Explicitly Modeling Hierarchically Heterogeneous Software Architectures in an Object-Oriented Formal Transformation System

THESIS
Darin L. Williams, Capt, USAF

AFIT/GCS/ENG/00M-25

20000815 176

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U. S. Government.

AFIT/GCS/ENG/00M-25

Explicitly Modeling Hierarchically Heterogeneous Software Architectures in an
Object-Oriented Formal Transformation System

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Systems

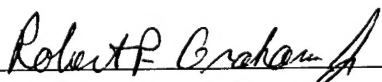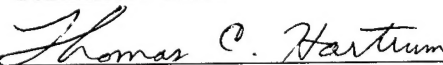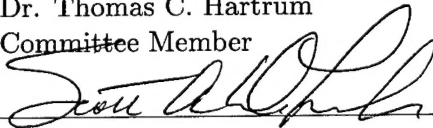Darin L. Williams, B.S.

Capt, USAF

March 2000

AFIT/GCS/ENG/00M-25

Explicitly Modeling Hierarchically Heterogeneous Software Architectures in an

Object-Oriented Formal Transformation System

Darin L. Williams, B.S.

Capt, USAF

Approved:

Robert P Graham Jr.   _9 MAR 2000_

Maj. Robert P. Graham Jr.   Date
Committee Chair

Thomas C. Hartrum   _9 Mar 2000_

Dr. Thomas C. Hartrum   Date
Committee Member

Scott A DeLoach   _9 Mar 2000_

Maj. Scott A. DeLoach   Date
Committee Member

Acknowledgements

Consider it pure joy, my brothers, whenever you face trials of many kinds, because you know that the testing of your faith develops perseverance. Perseverance must finish its work so that you may be mature and complete, not lacking anything.

James 1:2-3, NIV

As a child I often marveled at the beauty and luster of my father's high school class ring. I'm convinced that his inconspicuous display of personal academic accomplishment instilled in me at a very young age the value (and cost) of a good education. Therefore, I would like to take this opportunity to express to both of my parents my sincere gratitude for their diligence in raising me up with an appreciation for academic achievement. In addition, I wish to express my love and appreciation to Mariko, Zachary, and Ariel for catering to my every need and willingly relinquishing their right of access to their husband and father during these eighteen months so that I might complete this personal goal. I'd like to thank my academic advisor, Maj Robert Graham, for giving me the freedom to find my niche in the software engineering research community and for insisting on nothing short of my maximum effort in pursuing my research goals. I thank my fellow KBSE researchers—Thommo, Buck, Gary, Mike and Dave—for making this process more fun than it was probably ever intended to be. Above all else, I give full credit for this accomplishment to my ever present Lord and Savior, Jesus Christ.

Darin L. Williams

Table of Contents

List of Figures

List of Tables

AFIT/GCS/ENG/00M-25

Abstract

Formal software transformation systems are software development environments typified by the semi-automated application of a series of correctness-preserving transformations to formal data models. The range of software architectures such systems are capable of producing is often restricted by the limited ability to accept high-level design inputs as constraints on the transformation process. When architectural inputs are acceptable, often the modeling language excludes the explicit representation of architectural constructs and provides, at best, an extremely limited architectural analysis capability. This research defines a high-level taxonomy of software architectures and proposes a way to explicitly model a broad class of architectures by adapting the native object-oriented modeling language to the task. Using the AFIT Wide-Spectrum Object-Modeling Environment (AWSOME) as a proving ground, it demonstrates the ability to fully automate the transformation of an object-oriented analysis model to a non-hierarchically homogeneous, object-oriented architecture. Additionally, it demonstrates the ability to explicitly model the richer class of hierarchically heterogeneous software architectures in an object-oriented transformation system and to gain insight into the behavioral characteristics of such architectures by exporting them to an architectural interchange language for external analysis.

xi

Explicitly Modeling Hierarchically Heterogeneous Software Architectures in an

Object-Oriented Formal Transformation System

I. Introduction

A fundamental characteristic of modern software systems is complexity, and engineering such systems is a difficult business. *Automatic programming*—building a program to solve a problem from a statement of the problem—is one of many approaches touted by researchers as a potential solution to the inherent complexity of software development. In 1986, Frederick P. Brooks published a classic paper that cast doubt on automatic programming—along with several other promising technologies—as a potential silver bullet. He claimed that after 40 years of hype, he could scarcely conceive of the eventual realization of automatic programming [10:193–4]. Now, almost 15 years after their sentencing, advocates of automatic programming are alive and well, tooling away at its equally revolutionary cousin—formal transformation systems. The Knowledge-Based Software Engineering (KBSE) research group at the Air Force Institute of Technology (AFIT) is aggressively pursuing this approach through its development of the AFIT Wide-Spectrum Object Modeling Environment (AWSOME)[1]—an object-oriented, formal transformation system.

This work enriches that pursuit by integrating explicit architectural modeling into the formal, object-oriented, transformational process model. In briefly describing each of these process model descriptors (i.e., formal, object-oriented, and transformational), the following background section raises the hood of the transformational paradigm and lays bare the specific issues targeted by this research.

---

[1]AWSOME was originally called *AFIT*tool and was renamed during this research cycle at AFIT to reflect the expansion its underlying metamodel from a *design* language to a *wide-spectrum* (analysis and design) language.

1.1 Background

1.1.1 Formal Environments. In general, this research targets formal software engineering environments. Like any engineering discipline, software engineering is a problem-solving activity. The formality with which it is undertaken varies from organization to organization. Most software development organizations engage in the less formal approach, where specifications take on a prose format, designs and implementations are derived by hand, and systems are informally verified and validated against informal requirements and customer expectations. The benefits and detriments associated with an informal approach are discussed further in Chapter 2.

Formal methods, on the other hand, are mathematically based tools and techniques that greatly minimize the problems associated with less formal methods. Formal methods are characterized by mathematically rich languages, like Z, the precision of which facilitates unambiguous problem specification and increases the likelihood of correct solutions.

1.1.2 Object-Oriented Environments. More specifically, this research targets those formal environments that engage in object-oriented software development. Historically, software methodologies have over-emphasized either the data used in a problem environment or the functions performed on that data. By modeling real world entities, object-orientation attempts to resolve this bipolar arrangement by coupling the functions of interest with the data upon which they operate.

Rumbaugh's Object Modeling Technique (OMT) is the AFIT KBSE research group's object-oriented methodology of choice. They have applied various formalisms in specifying and representing the OMT's structural, dynamic, and functional models resulting in a formal object-oriented software development environment. While interesting in its own right, this description is yet incomplete. The semi-automated approach taken by the KBSE group leads to the third major descriptor—transformational software development.

1.1.3 Transformational Environments. Transformational environments, or *transformation systems*, are typified by the presence of formal data models that undergo a series of semi-automated, correctness-preserving transformations. These transformations

2
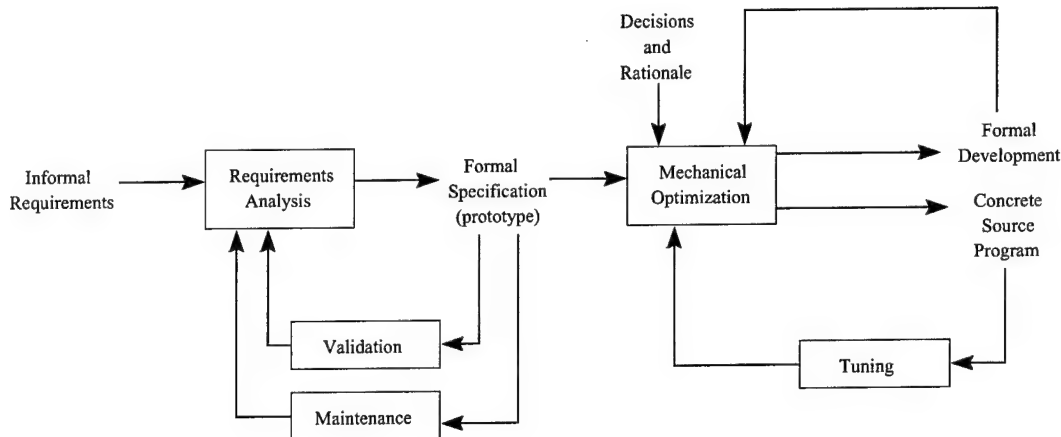
Figure 1.   The automation-based (transformation) paradigm.

progressively transform data from a high-level to a low-level of abstraction, i.e., from requirements specifications through design specifications to executable code [16]. Such systems were first described in detail by Balzer, Cheatham, and Green as providing for formal specification where the specification becomes the prototype, the prototype becomes the implementation, testing is eliminated, and maintenance is accomplished at the abstract specification level [4] (see Figure 1).

AWSOME is one implementation of just such a formal, object-oriented, transformation system. Although this research targets the general model, AWSOME provided a real environment to test its propositions. The propositions to be tested arose from various issues associated with yet another angle on object-oriented, formal transformation systems—their ability to explicitly model and manipulate a variety of software architectures. Before getting into those propositions and the issues they address, the next section offers a brief introduction to current issues in software architecture research.

1.1.4 Software Architecture.   Software architecture—the art and science of creating the architectural model, or high level design, of software systems—is one software development activity that, historically, has received a great deal of lip service, but very little real attention. That is beginning to change. As software development continues its evolution into an engineering discipline, it relies more and more on the rigorous application of its supporting technologies, one of which is software architecture. The argument for the

3
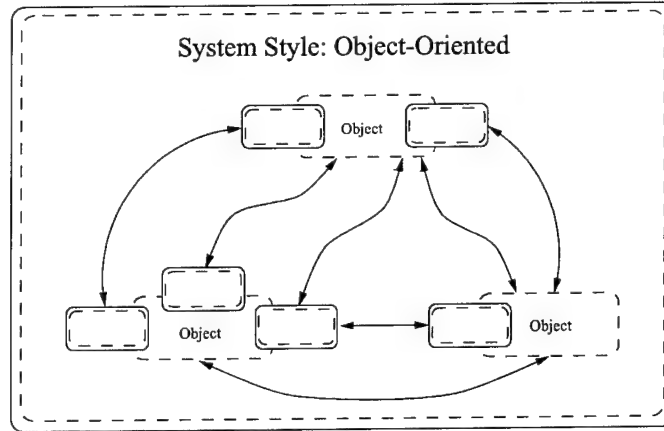
Figure 2.   An object-oriented software architecture.

importance of explicitly modeling the architectures of software systems during system design is covered in more detail in Chapter 2. For the moment, however, the most important issue is the nature of a software architecture.

Definitions for software architecture abound. Chapter 2 provides a definition suitable for the purposes of this thesis. In this introduction, however, suffice it to say that a software architecture is a recognizable collection of system components and the connectors between those components. There are many styles of software architecture, each offering its own connector and component types and its own topological and semantic constraints. One such style is the *object-oriented* style (see Figure 2). In an object-oriented architecture, the *objects* in the model are the architectural components and the *messages* passed between the objects, in the form of method calls, are the architectural connectors.

Often, however, software systems do not conform to one particular style. In fact, they exhibit what Bass, Clements, and Kazman call *hierarchically heterogeneous* software architectures [5:102]. This simply means that the architecture is often a hierarchical mixture of styles. For instance, in a distributed client-server architecture, the *client(s)* and *server(s)* are themselves components. They communicate via *socket* or remote procedure call (RPC) connectors. These components may, however, be further described as having object-oriented sub-architectures, whereby they are each comprised of one or more *object* components that communicate via *method call* connectors (see Figure 3). It is this tendency to build hierarchically heterogeneous architectures to solve even moderately complex

4
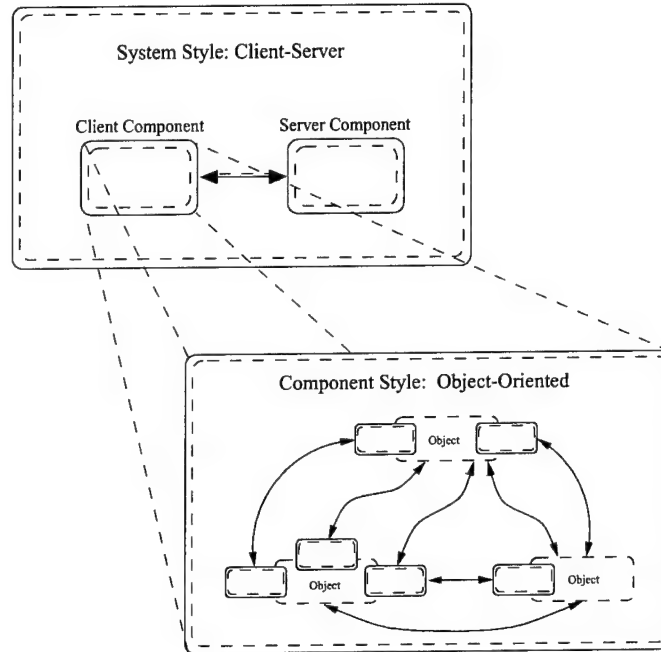
Figure 3.   A hierarchically heterogeneous software architecture.

problems, and the need to explicity and formally model these architectures , that leads to the problem addressed by this research.

1.2   Problem

Problem Statement: Many object-oriented, formal transformation systems are unable to explicitly model software architectures more complex than a network of objects that communicate via method calls. This research proposes a new and innovative approach to the high-level design or *architecture* of object-oriented systems that facilitates the explicit representation and extraction of hierarchically heterogeneous software architectures.

Specifically, this research addresses three problems in light of the previous discussion of hierarchically heterogeneous software architectures. First, object-oriented transformation methodologies only implicitly describe the architectures of the software systems they produce. In other words, there is no specific stage at which the engineer employs transformations that generate the architecture itself, nor are there explicit elements in the generated design model that equate to the semantic entities of software architectures (e.g., components, connectors, ports, roles, attachments, etc.). To be sure, any software

system produced by a transformation system—or any other methodology—*has* an architecture [5:24]. The issue is the level of architectural emphasis applied during its derivation and the degree to which the architecture is explicitly reflected in the design and used for preliminary evaluation of candidate solutions. Because an implicit architecture is only a shadow of the object-oriented design, any attempt to extract its composition can only do so by *inferentially* examining the elements of the low-level design and deciphering their architectural significance. Chapter 2 extends the argument for the importance of explicit software architecture modeling. Assuming the validity of that argument, a formal, object-oriented, transformation system must provide the means to explicitly represent architectural entities.

Second, a provision for explicitly modeling object-oriented architectures in an object-oriented transformation system is often not enough. As previously stated, most systems imposing any degree of complexity call for hierarchically heterogeneous architectures. In other words, it is not enough to explicitly model *object* components and *method call* connectors. A general purpose development environment must provide for a variety of architectural styles and the means to compose these styles hierarchically. Existing formal, object-oriented, transformation systems fall short of this capability by relying solely on the analyst's ability to capture the high-level design in the analysis model. This approach not only depends on the analyst's ability to recreate the essence of a particular style of architecture, but to do so in a way that changes the design-independent nature of the analysis model.

Finally, there is a tendency with systems that settle for implicit architectural modeling to extract software architectures from the analysis model rather than from the design model where the architectures actually reside. While analysis models provide an early reflection of the architecture in an object-oriented environment, it is not until the first high-level design decisions are made, e.g., objects are composed into components and events transformed into method call connectors, that the abstract requirements specification becomes a high-level design (i.e., an architecture). In fact, in an environment that recognizes the existence of a variety of architectural styles and the types of entities employed by those styles, inferring the architecture from the analysis model before it has been elaborated into a high-level design is tantamount to putting the proverbial "cart before the horse."

1.3 Initial Assessment of Past Effort

There has always been an interest in the architectures of software systems. But, not until the publication of *A Pattern Language* in 1977 by Christopher Alexander [1], the recognition of the applicability of patterns to software systems by the Pattern Languages of Program (PLoP) design community, and the work of others in the early 1990s did research in the area appreciably intensify. Most of the research has centered around frameworks, design patterns, or architectural styles. Very little has been done to integrate the accomplishments of these research groups with the work being done in automated software engineering.

Closer to home, in 1999, Penelope Noe provided a limited level of architectural support in the *AFIT*tool environment [22]. In her master's thesis, Noe recognized a useful relationship between the dynamic model of a formal, object-oriented software specification and the architecture of the target software system. Specifically, she provided a way to output the architecture of a system once the structural and dynamic models of that system had been specified. Unfortunately, architectures were only a peripheral concern within the context of her thesis; therefore, she afforded little attention to the issues of explicit architectural modeling and style-based software architectures. In addition, her approach is one example of looking for the right abstraction (the architecture) in the wrong place (the analysis model) and could only have been accomplished correctly in an environment devoid of richer architectural styles and compositional capabilities[2].

David Robinson broached the subject of software architectures from an agent-based software engineering perspective. As a member of AFIT's Agent Research Group (ARG), Robinson developed a formal language called AgDL (Agent Definition Language) that can be used to formally specify the agents in an agent-based software system. He validated the language by demonstrating its use with a variety of agent-based architectural styles (reactive, knowledge-based, planning, and Belief Desire Intention(BDI)). While providing an important capability within the agent research community, Robinson's coverage of ar-

---

[2]Note that in the *AFIT*tool environment to which she was accustomed, one could be sure that all classes and objects in the analysis model would become class and object components in the design, and all events in the analysis model would become method call connectors in the design.

chitecture modeling in his development of an agent specification language was too limited in scope to be used effectively in this research [26].

Finally, in his Master's thesis David Marsh demonstrated the ability to transform dynamic models in an object-oriented transformation system [20]. A secondary objective was to show that a system tooled to produce object-oriented software systems could be used to produce agent-based systems as well. Marsh's approach, however, was not a direct attempt to incorporate explicit architectural modeling into formal transformation systems. Rather, it adopted the design-enriched analysis model paradigm that results in an analysis model with, perhaps, more design information than would be desired by the software engineering purist. The result was that his approach—while providing significant contributions in terms of dynamic model transformation—was largely, if not completely, silent on the importance of explicitly modeling hierarchically heterogeneous software architectures as high-level design abstractions.

1.4 Scope

This research is primarily concerned with four architectural issues:

1. Explicitly modeling object-oriented software architectures in a formal object-oriented transformation system,

2. Representing hierarchically heterogeneous architectures in a formal, object-oriented transformation system,

3. Extending the transformational process model to include explicit architectural modeling, and

4. Exporting software architectures from formal, object-oriented design specifications to architectural interchange languages.

At the outset of this effort, the scope included various related issues such as eliciting and formally representing software quality attributes, creating a utility to facilitate semi-automated derivation of software architectures, and verifying conformance of software architectures to the architectural styles they implement. During the course of this effort,

8

they were eventually excluded from extensive consideration here and left for future research. They are, however, discussed throughout the thesis as they relate to the specific areas addressed.

Finally, this research targets only general purpose development environments. The author defines a general purpose environment as one that is geared toward the production of solutions to a variety of problems with no predisposition for a particular design or implementation and fully capable of producing solutions that exhibit a variety of architectural classes and styles. Many development environments specialize in the reproduction of specific architectural or design solutions to resolve problems. Such an approach appropriately leverages the knowledge about a particular genre of software designs in the resolution of problems meeting certain criteria. DeLoach's agentTool is one example of an environment that is tooled for the production of a specific class and style of architectures (i.e., agent-based systems). Such environments are not the target of this research. Rather, it provides a way, in a general purpose environment, to select and explicitly model one of many styles of archtitecture to solve a given problem.

To demonstrate the feasibility of the proposed methods for addressing the issues above, Chapter V incorporates them into AWSOME and exercises the new environment against a simple client-server software problem.

1.5   Document Overview

Chapter II reviews various research activities related to software architectures, formal methods, object orientation, and transformational software development. It also defines software architecture, establishes its importance as a key software engineering activity, and describes the AWSOME environment. Chapters III and IV present the author's approach to solving the problems described above. Chapter V implements the approach described in Chapters III and IV. The sixth and final chapter presents the results of the implementation, the author's conclusions based on those results, and specific recommendations for future research in areas related to the topic. The appendices provide additional information pertaining to the demonstration of much of the ideas in this thesis. Appendix A provides a formal Z specification for the example problem (**Bā′ bəl**) used throughout

this document. Appendix B provides an analysis model for $B\bar{a}'b\partial l$ in the AWSOME surface syntax (AWSOME is described in more detail in Chapter II). Appendix C and Appendix D provide an architectural model in AWSOME surface syntax and the equivalent Acme surface syntax for an object-oriented $B\bar{a}'b\partial l$ architecture, respectively. Finally, Appendix E and Appendix F provide equivalent AWSOME and Acme representations for an object-oriented client-server version of $B\bar{a}'b\partial l$.

II. Background

2.1 Introduction

In the community of applied sciences, software engineering is the new kid in town. In fact, use of the *engineering* label is rather premature and less than justifiable according to many in the field. It is clear, however, that software development is on the evolutionary path from craft to professional engineering discipline. The refinement and inculcation of software architecture as a fundamental activity within software development greatly facilitates that evolutionary process [28:5–14]. This chapter summarizes several key contributions to software architecture, reviews the latest trends in software development, and introduces an example transformational programming environment.

2.2 Software Architecture

A syntopical analysis of the subject reveals a great variety of recent contributions to the understanding and practice of software architecture—far too many for a comprehensive overview. Instead, those contributions related to

- the definition of software architecture as a software engineering activity,
- the justification of software architecture as an explicit process, and
- the codification of software architecture domain knowledge

are deemed especially valuable and presented below.

2.2.1 Software Architecture Defined. With increased research emphasis on software architecture, the question often arises: "what *is* a software architecture?" Is it simply "what the architect specifies in an architecture description" [17:83]? In their seminal work on the subject, *Software Architecture: Perspectives on an Emerging Discipline*, Mary Shaw and David Garlan define software architecture as

> ...the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns....a particular system is defined in terms of a collection of components and interactions among those components [28:1].

11

Roger Pressman, in the third edition of his text on software engineering, suggests a software architecture represents the *transition* between the requirements and design phases of the software lifecycle. It is accomplished by mapping each part of a software problem to one or more elements of the specified solution. He emphasizes the structuring of data and components as *the* primary architectural activity and states that the elements of a solution can be structured in a variety of ways [23:325-6]. Later, in his fourth edition, Pressman adopts the framework suggested by Shaw and Garlan, agreeing that software architecture is "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system" [24:351] [28]. Sommerville, while recognizing the absence of a "generally accepted process model for architectural design," identifies three activities commonly present in such models: system structuring, control modeling, and modular decomposition [30:226-7]. In their *Unified Software Development Process*, Jacobson, Booch, and Rumbaugh conclude that an architecture provides a suite of *views* of a software system—indeed, a abstract view of the "whole design"—that serves to communicate the "most significant and dynamic aspects of the system" [17:6].

The common thread running through these ideas on the nature of software architecture seems best summarized by Bass, Clements, and Kazman in *Software Architecture in Practice*. In this text, the authors build on the foundation laid by Shaw and Garlan and define software architecture as "...the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them" [5:23]. Figure 4 graphically depicts the essence of this definition.

Included in most definitions of architecture were the terms: component, connector, port, role, attachment, configuration, representation, and binding. The definitions for these terms given by Shaw and Garlan [28] are provided here and used throughout this report.

Component. The loci of computation, components represent the primary building blocks of an architecture.

Connector. Defines a path of interaction between components.
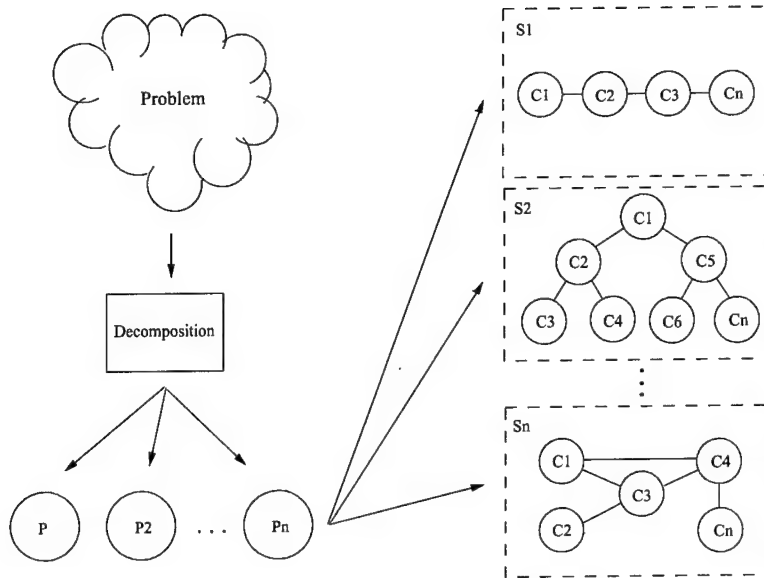
Figure 4.   Architectural structures.

Port. Represents a component's points of interaction with its environment.

Role. Represents a participant in an interaction; a set of roles makes up the interface for a connector.

Attachment. Represents a link between a component's port and a connector's role.

Configuration. Defines a topology of components and connectors.

Representation. A way to represent the "contents" of a component or connector. Representations enable the hierarchical composition of architectures.

Binding. In a hierarchical architecture, bindings provide the links between sub-component ports in a representation with the ports of the parent component (i.e., the component being defined by the representation).

2.2.1.1   The Rational Unified Process.   Despite the growing consensus on how to define software architecture, there is by no means a well understood process model for formally deriving an architecture for a new system or examining a preexisting one. Jacobson, et al., present a relatively complete software development process model that

13
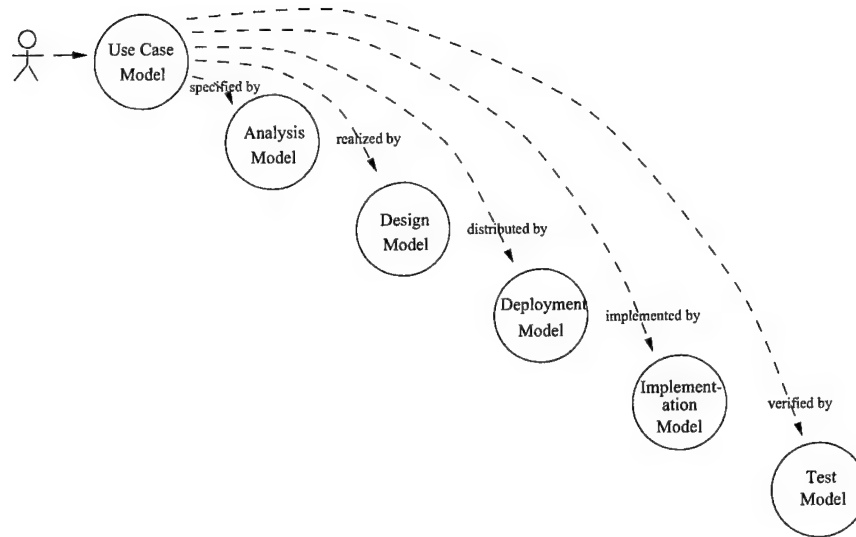
Figure 5.   Models of the Unified Process.

includes semi-formal software architecture modeling [17]. Their *Unified Process*[1] is characterized, in fact, as being architecture-centric and use-case driven. As depicted in Figure 5, the Unified Process is based on a series of interdependent system models beginning with the *use-case model* and ending with the *test model*. While communicating the notion of interdependent models, this diagram is an oversimplification. There are, in fact, many other dependencies between the models of the system [17:10].

One interesting aspect of the Unified Process is the idea that use cases—used primarily to capture the *functional* requirements for a software system—are the primary drivers of the software architecture. While this may seem plausible, it constitutes a significant deviation from the assertion by Bass, et al., that software architectures are primarily driven by software quality attributes, not functional requirements. To be sure, the authors of the Unified Process certainly include non-functional requirements (i.e., quality attributes) as architectural drivers, but they are relegated to a position of minimal influence in that process model. Bass, et al., took a different approach in *Software Architecture in Practice* by describing software architectural modeling in the context of the *architectural business cycle* (ABC).

---

[1]The Rational Unified Process is the latest version of a model that has been evolving since 1987. As recently as 1997, it was known as the Rational Objectory Process.

14
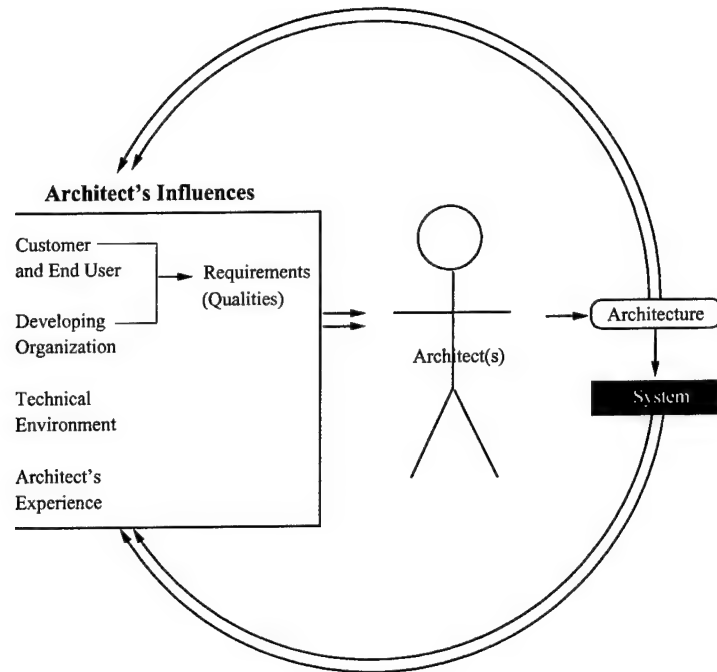
Figure 6.   The Architectural Business Cycle.

2.2.1.2   The Architectural Business Cycle.   According to Bass, Clements, and Kazman, software architectures are not derived from functional requirements. They make the point by suggesting that two different architects working in isolation from the same requirements specification will likely produce two completely different architectures. Accordingly, there must be something other than functional requirements driving software architectures. This *something,* according to the authors, is the technical, business, and social influences emanating from the surrounding environment. The primary factors influencing the architecture are system stakeholders (external and internal), the technical environment in which the system is engineered (current tools, methods, and methodologies), and the architect's personal experience. Figure 6 illustrates this influential relationship between the architect and his or her environment. It is equally important to note the existence of a feedback loop from the architecture and the system itself back to the environment. This reinforces the notion of a complete cycle where the environment influences the creation of a new architecture, and the new architecture, in turn, influences the environment [5:11].

15

Quality Attribute	Architectural Issues
Performance	Intercomponent communication
Security	Specialized components
Availability	Redundant components
Usability	Achieving proper information flow
Modifiability	Modularization; encapsulation
Portability	Layers
Reusability	Decoupling
Integrability	Component interfaces
Testability	Modularization; encapsulation

Table 1. Quality attributes and architectural issues.

Arguably the most important point made by the authors is that software quality attributes (i.e., non-functional requirements or "-ilities") drive a specific architectural solution to a problem. At the same time, quality attributes often conflict, so decisions must be made regarding the relative importance of those attributes and the architectural styles or patterns that are likely to support them. What are these attributes? Table 1 lists the quality attributes discussed by the authors and the issues that must be addressed by the architecture [5:86].

Their discussion of software architectures quickly moves from one of environmental influences and quality attributes to architectural styles. The significance of a catalog of architectural styles, their important relationship to design patterns, and the movement towards codification of this domain knowledge warrants separate treatment of the subject. But, first, a justification for architecting software systems is presented.

2.2.2 Software Architecture Justified. So, why trouble oneself with all this overhead? After all, software developers have managed to avoid the issue of architectures by ignoring its importance (at worst) or implicitly modeling them (at best). This challenge to justify the apparent overhead of software architectural modeling is a fair one, the response to which has been clearly addressed. The benefits of explicit software architecture modeling during the early stages of design fall into six general categories: improved organization, communication, and understanding; early analysis; controlled evolution; and standardization.

16

1. *Software architectures facilitate improved* **organization** *of major software development efforts.* Modern software systems, especially the ones for which there is an architectural interest, are typically very large systems requiring the combined efforts of many software specialists. In fact, it is entirely possible to have hundreds, even thousands, of people involved in a major software intensive development program [10:31]. The architectural breakdown of a large system enables clean assignment of work to the often geographically distributed teams and individuals assigned to the project. With the interfaces to these subsystems clearly defined and controlled by system architects, small groups are free to focus on their specific components, with relatively little regard for the parallel efforts of others [17:63]. Large systems must inevitably be partitioned for development—without an overarching architecture to guide this process, an ad hoc partitioning scheme can generate more problems that it seeks to resolve.

2. *Software architectures aid* **communication** *among stakeholders.* There are many different perspectives that come to bear on a software system development effort, both internal and external to the developing organization. The customer is concerned with schedule and budget issues, while the users are more interested in system functionality. Managers worry about controlling large project team interaction, while implementers concern themselves with data structures, algorithms, and programming languages. Everyone has a stake in the operation and, therefore, an interest in the key decisions made early in the construction phase that will affect the software system long-term. With increased involvement, communication overhead is introduced [17:63]. The software architecture created early in the design process aids the communication process by providing an abstract solution that is easily communicated to all stakeholders [5:28]. Without an architectural abstraction, the key players are left to examine the problem specification or the solution design. The specification is understandable by customers and users, but addresses the problem space, not the solution space. The design targets the solution, but is typically too far down the abstraction chain (i.e., too low-level) to be meaningful to all but the developers.

17

3. *Software architectures promote greater **understanding** of complex software systems.* As previously stated, modern software systems are typically very complex and difficult to grasp in their entirety by most individuals. In fact, Brooks suggests that software systems are the most complex structures engineered by human beings— orders of magnitude more complex than even the digital computers on which they operate [10:182-3]. Architectures provide a high level partitioning of a solution, using a language or presentation mechanism that all can understand, to address this complexity. A comparable level of understanding is much more difficult to achieve once developers engage the solution at a lower level of abstraction using tools and languages best suited to the task.

4. *Software architectures enable early **analysis** of the efficacy of a proposed software solution.* The software system architecture is the earliest opportunity for stakeholders in a software development effort to evaluate the ability of a proposed solution to adequately solve a specified problem [5:28]. Two related principles help illustrate this benefit. The *cost-error* principle states that the earlier errors are detected in a process, the less expensive it will be to correct the errors. The *Albert Einstein principle* states that the proper formulation of the problem is even more essential than its solution. Analysis of a good software architecture gives us the ability to detect errors in the target solution to a problem earlier than would be possible without an available architectural abstraction. Additionally, it is the software architecture that allows us to provide a proper formulation of the problem before engaging the finer details of algorithm and data structure design [29].

5. *Software architectures help maintain the conceptual **integrity** of evolving software systems.* Software systems are in a constant state of change, both during development and later during operation and maintenance [17:64]. These systems must be designed in such a way that they can be easily adapted to changing requirements and environments. Systems constructed around good architectures have this resilience— the architecture serving as the *conscience* of the system [21]. As early as 1975, Frederick P. Brooks, Jr. addressed the necessity of software system architectures in his classical book, *The Mythical Man-Month*. In this masterful treatise on managing

complex programs, Brooks opines that *the* single most important consideration when designing a software system is the *conceptual integrity* of the system. A system with conceptual integrity is one whose parts—whether developed as part of the original system or appended later—all seamlessly integrate with, and contribute to, the original system design. Brooks offers the Reims Cathedral as a wonderful example of conceptual integrity in action. Despite the involvement of eight generations of designers and builders, the cathedral exhibits a magnificently singular architecture—a testimony to the selfless adherence of subsequent designers to the overarching concept espoused by Jean d'Orbais [10:41–2]. So it should be with software that, in some cases, has been in operation and maintenance for over 40 years.

6. *Software architectures simplify the* **reuse** *of proven software design patterns.* Standardization and *componentization* (component-based construction) are hallmarks of a mature engineering discipline. Architectures foster the development and repeated use of well-known domain patterns and components, greatly decreasing the time and cost normally associated with the "creative" alternative and increasing predictability of the results. Low-level design patterns have been, in fact, a hot topic for some time. Selecting a compatible group of patterns, however, and composing them in reasonable ways to solve complex computing problems is not well established. Architectural modeling can provide the higher level packaging of reusable design patterns into components that can be arranged in proven ways.

Clearly, there is great value in architecting a software system, and many have taken advantage of the benefits provided. However, to move closer to an engineering paradigm, where practitioners reuse proven solutions rather than create new ones, the reusable artifacts and, in formal environments, the knowledge necessary to effectively implement the artifacts must be codified.

2.2.3 Software Architecture Codified. As suggested in the introduction, codification of software architectural knowledge is an important step in the evolution of software development from a craft to an engineering discipline. For codification to occur, a formal, descriptive language capable of describing the software architecture domain and specific

19

software architectures must be selected. The software architecture domain can be viewed from three distinct, yet related perspectives: architectural *styles*, architectural [design] *patterns*, and architectural *frameworks*. The obvious question is, what is the difference between a style, a pattern, and a framework? Essentially, a style is a language for describing a family of architectural patterns, and a pattern is a generalization of a framework. The three constructs are discussed in more detail in the following paragraphs.

2.2.3.1 Architectural Styles. For as long as designers have been constructing software systems, architectural styles have been around. For the most part, these styles were no more than idiomatic descriptions of a particular method of construction. For example, the often occurring configuration of client programs subscribing to the services of server programs via procedure calls (or some other interaction mechanism) became commonly referred to as a *client-server* architecture. When used by experienced designers, this simple description says much about the general structure of the components and connections in the system while avoiding unnecessary detail. Furthermore, systems are rarely pure instantiations of a particular style—rather, they usually reflect a useful composition of various styles. These are referred to as heterogeneous architectures [28:19–32].

Several attempts have been made to categorize the most common styles. Shaw and Garlan defined the categorized list shown in Table 2. In addition to a descriptive name and a graphical representation, the authors recommended—and in some cases provided—informal definitions of architectural styles in terms of their

- design vocabularies (types of components and connectors),
- structural patterns,
- underlying computational models,
- essential invariants,
- common uses (examples),
- advantages and disadvantages, and
- specializations.

Bass, Clements, and Kazman built on the styles offered by Shaw and Garlan. With the exception of minor name changes, they focused on characterizing the styles in greater

Architectural Styles
Dataflow systems
Batch sequential
Pipes and filters
Call-and-return systems
Main program and subroutine
Object-oriented systems
Hierarchical layers
Independent components
Communicating processes
Event systems
Data-centered systems (repositories)
Databases
Hypertext systems
Blackboards
Virtual machines
Interpreters
Rule-based systems

Table 2.   A catalog of architectural styles.

detail. In particular, they defined the constituent parts, control issues, data issues, and control and data interactions associated with each style. Fig 7 shows the definition of several styles as recorded by Bass, et al. Their complete list can be found at [5:108–11].

Two of the *call-and-return* styles were of particular interest during this research—the *objects* (or object-oriented) style and the *call-based client-server* (or simply client-server) style [5] [28]. These two styles are highlighted here and used in later chapters to illustrate the ideas presented.

- **Object-oriented style.**

  - *Components*: Managers (objects)

  - *Connectors*: Dynamic calls (method calls)

  - *Semantic Constraints*: An object must preserve the integrity of its representation and the representation is hidden from other objects.

  - *Topological Constraints*: Data topology is arbitrary; control flow topology is arbitrary; data and control flows in same direction.

	Constituent Parts			Control Issues		Data Issues				Control/Data Issues	
Style	Components	Connectors	Topology	Synchronicity	Binding time	Topology	Continuity	Mode	Binding time	Isomorphic Shapes	Flow directions
Data-centered: dominated by a complex central data store, manipulated by independent computations										**Type of reasoning:** Data integrity	
Repository	Memory, computations	Queries	Star	Asynchronous opportunistic	write time	Star	Sporadic low volume	Shared passed	write time	Possibly	If isomorphic, opposite
Blackboard	Memory, computations	Direct access	Star	Asynchronous opportunistic	write time	Star	Sporadic low volume	Shared multicast	write time	No	n/a
...											
Call-and-return: dominated by order of computation, usually with single thread of control										**Type of reasoning:** Hierarchy (local reasoning)	
Main program/ subroutine	Procedures, Data	Procedure calls	Hierarchical	Sequential	write time compile time	arbitrary	Sporadic low volume	Passed, shared	write time compile time run time	No	n/a
Layered	Various	Various	Hierarchical	Any	Any	Hierarchical	Sporadic low volume continuous	Any	write time compile time run time invocation time	Often	Same or opposite
...											

Figure 7.   Defining and classifying architectural styles.

22

- **Client-server style.**

  - *Components*: Clients, Servers

  - *Connectors*: Calls or Remote Procedure Calls (RPC)

  - *Semantic Constraints*: Servers provide black-box services; Clients request services provided by Servers.

  - *Topological Constraints*: Data topology is star; control topology is star; data and control flows in opposite directions.

While architectural styles are useful in the sense that they provide easily recognizable schemes for arranging the components in a software system, some suggest they lack the level of practical reuse necessary to realize the benefits described in Section 2.2.2. Styles specify building block types and establish constraints on how a system can be defined, but essentially require the designer to construct the system in its entirety [21]. Architectural *patterns* may bridge the gap between abstract architectural styles and complete software system architectures.

2.2.3.2 Architectural Patterns.   Reusable design patterns in general have captured the imaginations of software professionals for several years. The Pattern Languages of Programs (PLoP) community sparked serious dialog on the subject in the early 1990's by calling for the launch of a new literature dedicated to patterns in software. The renewed interest in reusable software design patterns came about primarily as a result of the work of the building architect Christopher Alexander [11:2]. In defining design patterns, Alexander contends

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice [1:X].

PLoP cohorts recognized a growing problem with existing software engineering literature: a penchant for the discovery and development of new and exciting solutions—regardless of their utility—while ignoring the host of existing design patterns. Their efforts have resulted in the ongoing PLoP conferences and volumes of useful pattern-related literature [8].

23

In *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson, and Vlissides constructed a catalog of reusable software design patterns. Based on the purpose for each pattern, they assign meaningful names to facilitate learning and communication by practitioners (e.g., Abstract Factory, Facade, Interpreter, Visitor, etc.). The text contains a complete definition of each of the twenty or so patterns deemed most useful by the authors. In defining the patterns, they suggest that a complete design pattern definition includes a pattern name, a description of the problem solved by the pattern, a description of the pattern solution, and the consequences one can expect when applying the pattern (e.g., space and time trade-offs) [11:3].

2.2.3.3 Architectural Frameworks. Indeed, software design patterns are useful and offer much in the way of design reusability, but the most common patterns don't always apply when dealing with broader architectural issues. In fact, in their introduction to Part 1 of *Pattern Languages of Program Design*, Coplien and Schmidt describe a specialization of design patterns, called *frameworks*, that addresses these higher level issues. Frameworks attempt to capture the behavior of commonly occurring abstractions in an application domain—they are essentially "semicomplete" applications. The framework is *instantiated* much like an object-oriented class provided by a class library, but there is a significant difference in run-time flow control between the services offered by reusable frameworks and class libraries. When reusing classes from class libraries, the application code maintains control by determining what methods to invoke and when to invoke them. Frameworks, on the other hand, offer the ability to reuse the flow control modeled by the framework. Instead of writing application code to call reusable objects, one writes code that gets called by reusable frameworks. Frameworks are a careful balance of variability and stability. In other words, framework designers carefully weigh the variability required to enable "millions" of different instantiations of the framework against the stability required for confident reliance upon the reusable behavior offered by the pattern [8:1–5].

Codification of architectural knowledge in the form of styles, patterns, and frameworks is a necessary step in the path to effective and efficient generation of software system

24

architectures. Formalization of these structures will pave the way for incorporating explicit software architectural modeling in a formal transformation paradigm.

2.2.4 Architecture Description Languages.   Software architectures, and the styles, patterns, and frameworks used to generate them in a transformational environment must be described using a formal modeling language. Many such languages have been developed specifically for the task—they are generally called architecture description languages (ADLs).

Wright, an ADL developed by Robert Allen [2] as part of his Ph.D. dissertation, was built around the notions of *components, connectors,* and *configurations.* The language was heavily influenced by the Communicating Sequential Processes (CSP) language and is geared toward providing not only the capability to describe an architecture, but also the ability to analyze it. Aesop, developed by the ABLE Project at Carnegie Mellon University, is a system for generating style-oriented architectural design environments [12]. ACME is more an architecture description interchange language than an ADL [13]. It attempts to facilitate the interchange of architectural descriptions written in different languages using a common interfacing language. Other, more general purpose languages have also been used to formally specify software architectures. Shaw and Garlan demonstrate the use of Z to specify architectures [28][130–142]. Table 3 compares the use of general purpose formal languages to the use of ADLs for architectural specification.

2.3 Software Development Trends

The face of software development has evolved significantly since the late 1960s. Two specific trends are worth reviewing prior to broaching the subject of architectural modeling in a formal software transformation environment. The shift from a *structured* to an *object-oriented* paradigm across the industry and the gradual adoption of formal methods by many software development organizations are of particular interest.

2.3.1 Object-Orientation.   Regardless of the methodology used (e.g., structured, object-oriented, etc.), developing a software system typically involves specifying a series of

Technique	Advantages	Disadvantages
Formal methods	- provides for system property analysis - provides for verification of implementation - improves communication of system design	- a non-trivial undertaking - requires definition from first principles for each new description for each new description - reuse based on previous experience; lacks common framework for patterns and abstractions - each developer must invent own models and tools
Structure-based ADLs	- permit exposure/definition of systems - well suited to real system description - explicit modeling facilitates traceability - can efficiently generate reliable systems	- lack direct element property specification; limits analytical leverage - do little to highlight properties of target systems for analysis - limited range of systems that can be modeled in any particular language

Table 3.   The status quo in modeling software architectures.

abstract models[2] of the system based on the customer's needs (see Figure 8). Each model is a slightly less abstract view of the target system than its predecessor. The requirements model is the highest level abstraction, telling only *what* the system must do in order to be successful. Using the requirements model, systems analysts and designers create a lower-level abstraction called the design model. This model details *how* the system will implement the requirements. Programmers (i.e., implementers, coders) create the implementation model by coding the design using one or more programming languages suited to the task (e.g., Ada, C++, Java). Programmers provide this implementation model as input to compilers that translate the language-specific code into machine readable instructions—the execution model.

Traditionally, to derive the set of models for a system, developers focused on the functional and behavioral aspects of the problem domain and structured software systems

---

[2]The term *model* is preferred over *specification* in this instance, because the interest is in *every* abstraction of the target system up to, and including, the executable code itself. "Model" seems more appropriate since the source and object code are rarely considered specifications.
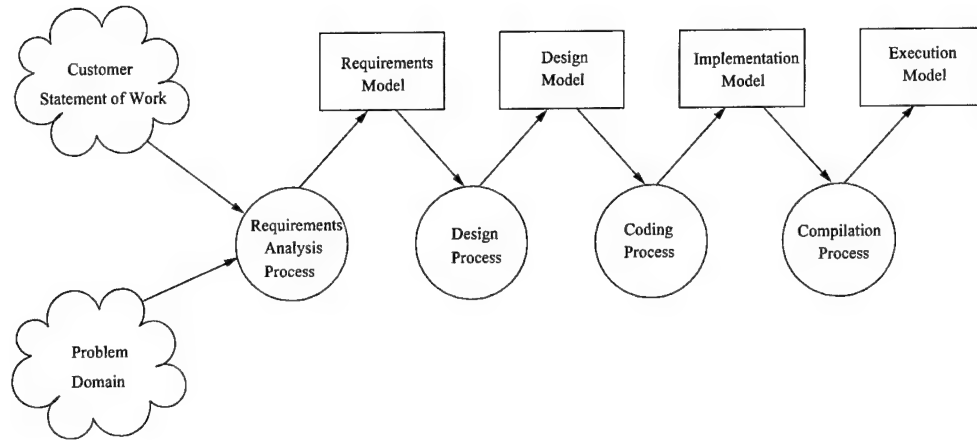
Figure 8.   Software modeling.

accordingly. Following this approach, the system is viewed as a hierarchy or network of interrelated processes. From the late 1960s to the mid 1980s, this *structured* paradigm drew much attention and evolved into what Pressman calls an "amalgam" of methods and techniques. By the mid 1980s, however, the new *object-oriented* approach had gained a foothold, and by the early 1990s, the structured approach had relinquished its hold on the majority of developers building new software systems [24:207,396–397].

Using object orientation, developers focus on the *objects* inherent to the problem domain, rather than the processes. In other words, they key on the *nouns* in a problem statement rather than the *verbs*. Now the system is viewed as a set of objects that communicate via message passing. Each object is associated with a *class*. A class is essentially a template, and an object is an instantiation of some template. The class definition identifies the operations accessible to the class (via methods[3]) and the data items (i.e., attributes) encapsulated by the class; this structure is depicted in Figure 9. Every object of a particular class has its own set of instance variables and methods. Many texts offer a much more comprehensive discussion of object-oriented technology; the interested reader is referred to Rumbaugh's text on object-oriented modeling for more information [27].

---

[3] A subtle difference between an *operation* and a *method* is that an operation defines, in general terms, the purpose of an action while a method implements the operation for a particular class. More will be said about this in the discussion of polymorphism.

```
+-----------------------------+
|          aircraft           |
+-----------------------------+
| id_number : string          |
| speed : integer             |
| heading : integer           |
| altitude : integer          |
+-----------------------------+
| get_number() : string       |
| get_speed() : integer       |
| set_speed(integer)          |
+-----------------------------+
```

Figure 9.   A typical *class* definition.

2.3.2 Formal Methods.   Formal methods are mathematically based techniques for specifying, verifying, and/or synthesizing a software system [6]. Traditionally, the emphasis has been on using natural language (e.g., English prose) or semi-formal modeling tools[4] and techniques to specify systems. Disproportionate use of natural language and semi-formal tools often results in specifications that are ambiguous, inconsistent, and incomplete. Formal methods, whether used sparingly or exclusively, serve to diminish these problems. In fact, Wordsworth defines *formalization* as "the process of making a vague notion precise" [34:5]. The formal methods designed to resolve these problems have held the attention of academia since the late 1960s, but have failed to gain widespread application in industry. Albert Einstein seemingly addressed the problem when he stated, "Why does this magnificent applied science which saves work and makes life easier bring us so little happiness? The simple answer runs: because we have not yet learned to make sensible use of it" [6]. He was right—not only is the shift to formal methods a significant change in the way we think about software, it requires a high degree of training and experience to be done effectively; a degree to which most organizations have been unable to commit.

---

[4]Many "semi-formal" tools have been developed to aid in system specification, verification, and synthesis (e.g., Rumbaugh's Object Modeling Technique (OMT), ERWin, Model Mart, Visual Studio Modeler, etc.); however, these tools typically do not provide a level of mathematical precision—and the associated benefits—characteristic of their formal cousins.

Nevertheless, the many advances made by researchers and practitioners alike promise to facilitate greater understanding and use of formal methods.

2.3.2.1 Formal Specification. A system specification identifies the properties of a target system at a particular level of abstraction. Formal specifications capture the desired what, when, and how of the system using specification languages with well-defined syntax and semantics. Specification languages can be model-based, property-based, sequential, concurrent, or a combination of these attributes. Model-based languages are used to build systems using mathematical constructs such as sets, relations, and functions. Pre- and post-conditions are used to specify the operations within the system. Property-based languages use axiomatic or algebraic notations to define the minimal properties of a system. Sequential languages are applied when the system is characterized by a single thread of operation while concurrent languages are used when dealing with multiple concurrent processes [14].

Z[5] is a widely used (relatively speaking), model-based, sequential, specification language. Z is of particular interest to this thesis, because it is the formal specification language of choice within the Knowledge Based Software Engineering (KBSE) Group at the Air Force Institute of Technology (AFIT). Z specifications consist of type declarations and schemas. Type declarations come in three varieties: *built in*, *basic*, and *free* [15:449]. The *built in* type consists of the set of integers denoted $\mathbb{Z}$. *Basic* types are defined by the specification and are usually unique to the application being specified. For example, to specify a FILTER type, the following notation is used: [FILTER]. This type represents the set of all filters. *Free* types are essentially enumerated types and are recorded in the following way: Component_Type ::= filter | repository | process | procedure. Attributes of type Component_Type, therefore, can only accept the values filter, repository, process, or procedure[6].

---

[5] Z is correctly pronounced "zed," indicative of its origination by J.M. Spivey of the United Kingdom.
[6] This enumerated Component_Type is included here only for illustrative purposes and is not meant enumerate all known types of architectural components

29

2.3.2.2 Formal Verification. Formal software verification is the method (or methods) used to ensure that a software system satisfies its specification. In order to perform formal verification, there must exist a formal specification and a formal semantics for the programming language used to develop the system. It is the comparison of these two formal artifacts that makes formal verification possible [14].

There are various methods of formal verification. Two of the more prevalent methods are *model checking* and *theorem proving*. Model checking is primarily used in hardware and protocol verification. This approach attempts to model the system and specification as finite state machines (FSMs) and perform automated comparisons of these FSMs. There are some difficulties related to this approach and it is not widely used in software verification.

Theorem proving, on the other hand, is more prevalent in the software community. Using this method, systems and specifications are modeled as logical formulae. Assertions are made about a program at certain points in the program. The objective is to verify (prove) that the assertion (or state) at a particular point in the program, coupled with the effects of the code at that point, imply the next assertion. Usually, this approach is applied to program fragments rather than an overall program—an approach derived from Floyd's Method of Inductive Assertions. Theorem proving can be accomplished both manually and automatically by *theorem provers*. The manual approach is error prone, but automated theorem provers are difficult to implement. In fact, the verification process can often become more complicated than development itself [14].

2.3.2.3 Formal Synthesis. *Synthesis* is the term used to describe the process of actually building the code for a system (i.e., development). By using formal (i.e., mathematically based) methods that have been proven correct to synthesize software systems, it is possible to build correct software without having to apply separate verification techniques [14]. In fact, having the methods in a formal format facilitates automation (to a large degree) of the synthesis process. In this way, programs are developed by applying a series of automated transformations to formal specifications. This approach to software development is commonly called the *transformational* approach. Figure 10 shows a model

30
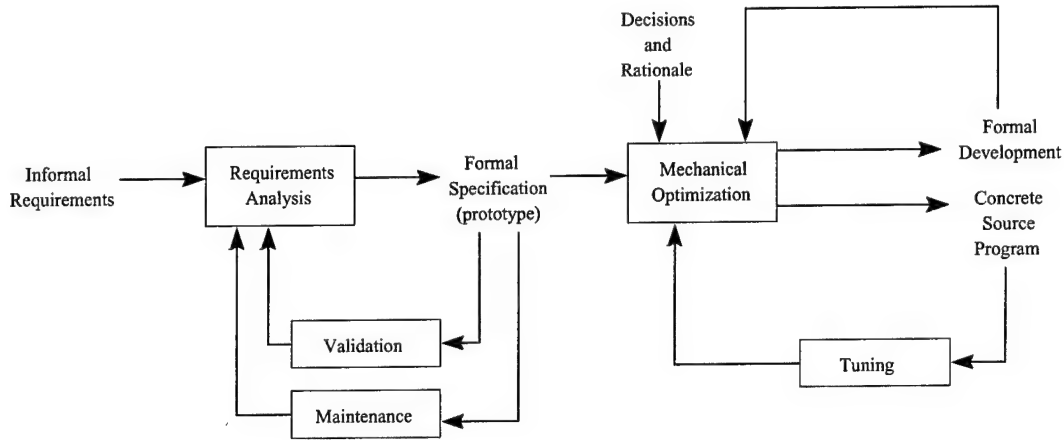
Figure 10.   The automation-based (transformation) paradigm.

for the transformational approach espoused by Balzer, Cheatham, and Green [4]. As shown in the diagram, software maintenance in a semi-automated transformation environment is shifted from the implementation model (i.e., the source code) to the more abstract formal specification. With the exception of key decisions and their rationale provided by software engineers, the automated transforms take control of the optimization and code generation functions.

There is a great deal more to say about formal methods in general and Z specifications in particular in regards to software engineering. To gain a deeper appreciation of formal methods, see the related articles by Bowen, Hinchey, and Jones [6, 7, 18, 19, 33]. While Grassman and Tremblay introduce the topic of Z specification in the context of discrete mathematics [15], Spivey, Woodcock, Loomes, and Davies present the latest techniques in Z specification based software engineering [31, 32, 34].

2.4   A Model Environment: AWSOME

The KBSE Group in the Department of Electrical and Computer Engineering at AFIT has been researching formal transformational software development for several years. To support and demonstrate the capabilities explored through their research, the group has evolved a formal, object-oriented, semi-automated, transformation system called AW-SOME.
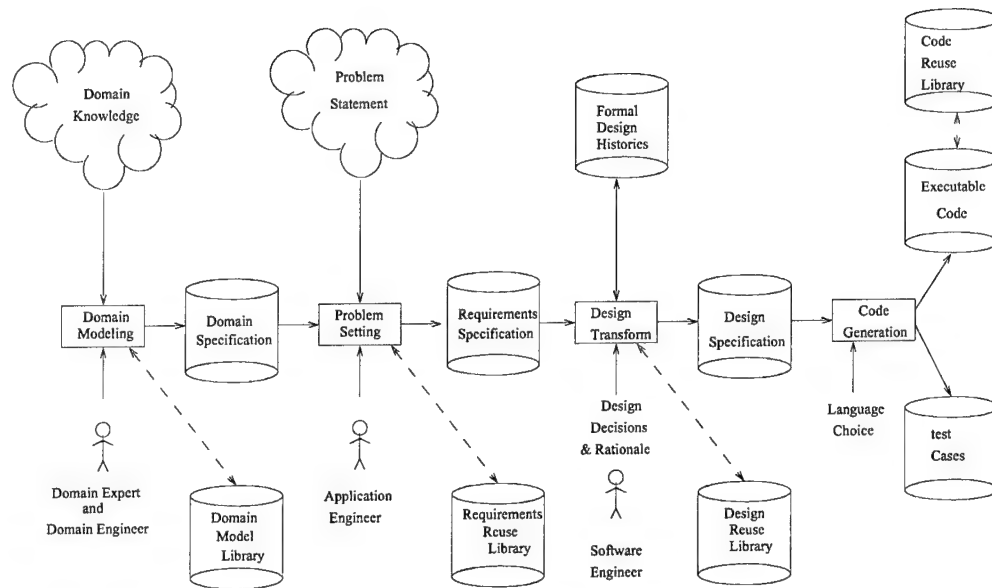
Figure 11. The AWSOME batch sequential architecture.

2.4.1 An Architectural Perspective. Although AWSOME is still the focus of research, it sufficiently demonstrates the real possibilities imagined by Balzer et al. Figure 11 depicts the batch sequential architecture of AWSOME. Each major process in the system manipulates its own data structure(s). For example, the problem setting process takes as input a model of the problem domain and produces a formal problem specification. The design transform process, in turn, manipulates the formal specification creating a design model.

A more recent variation on the AWSOME architecture recognizes the overlap in these data structures and integrates them into a centralized repository [9]; thus, AWSOME is currently represented as a data-centered repository architecture (see Figure 12). Generally, AWSOME can be viewed as a series of semi-automated processes that act on formal software system models. The specific processes are described by Hartrum [16:1–2] and summarized below:

Domain modeling: the application of object modeling techniques to the entire problem *domain*, perhaps independent of any particular application development effort, in order to capture and represent domain knowledge in a formal domain model.

32
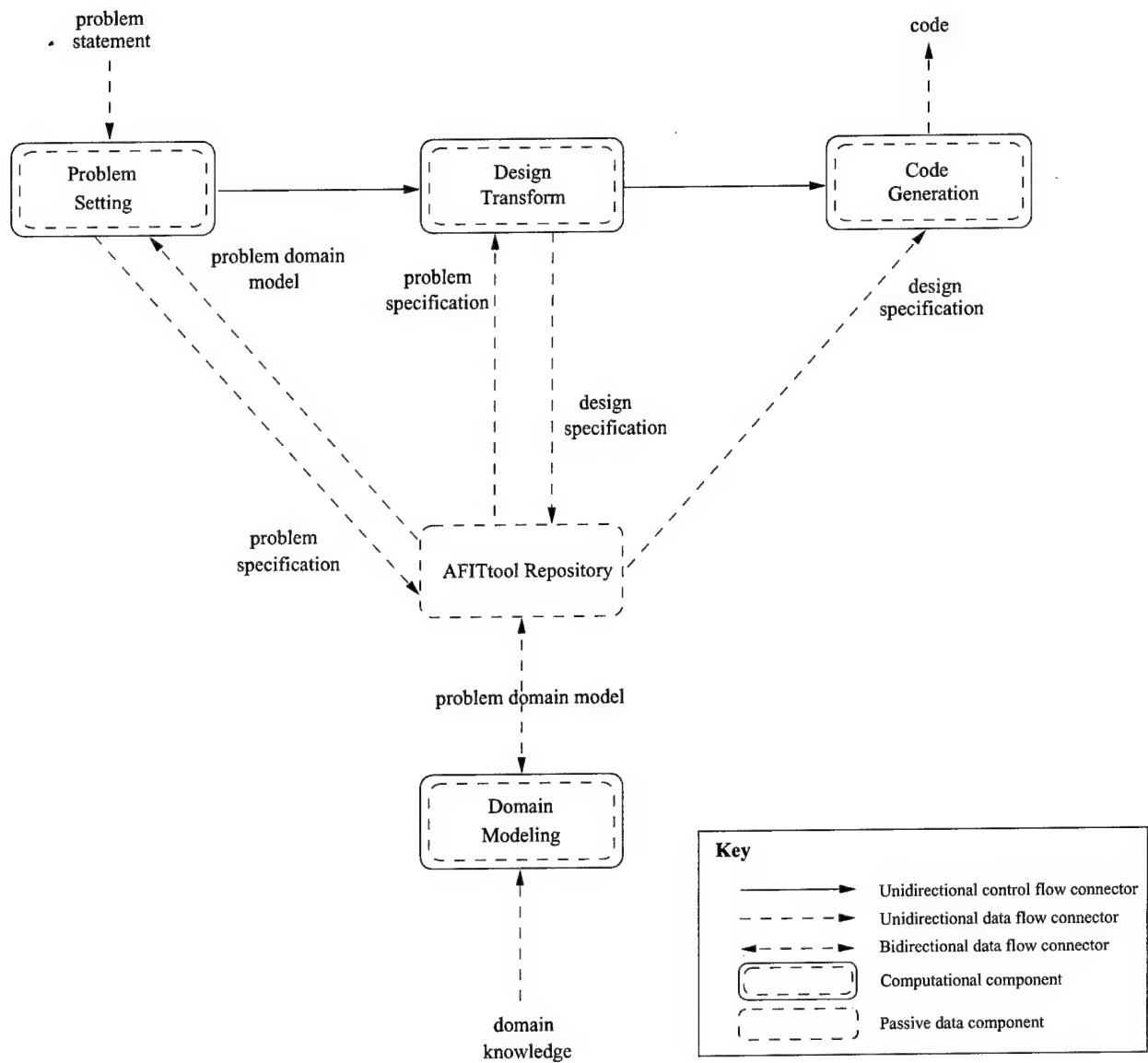
Figure 12. The AWSOME data-centered repository architecture.

Problem setting: the process of modeling a specific domain *application* by extracting relevant artifacts from the formal domain model and supplementing these with problem-specific entities, resulting in a formal object-oriented requirements specification[7].

Design transform: the semi-automated application of correctness-preserving transforms to the formal requirements specification, resulting in a formal object-oriented design specification. This specification represents the first tangible model of the solution (i.e., the target system).

Code generation: the interactive process of programming language selection and semi-automated transformation of the formal design specification into executable code.

Without data to process, the system would be incomplete. This is the role of the system models discussed in the following section.

2.4.2 The System Models. The system models (i.e., formal domain model, formal specification, and design specification) are stored and manipulated as abstract syntax trees (AST), that constitute the principal output of the transformation system. The domain model and formal specification share a common metamodel; that is, the models contain the same type of information with an identical abstract syntax. The primary difference between the two is one of scope. While the *domain* model contains knowledge about the entire problem domain, the formal specification contains only those classes relevant to the specific problem being studied. The specification encompasses three different views of the problem: the structural, functional, and dynamic views[8].

The design specification, on the other hand, models the solution. It contains lower-level detail sufficient for deriving specific data structures and algorithms required to support

---

[7]Since the formal domain model may not provide *all* the knowledge necessary to completely specify an application, the domain model harvesting process typically provides a mechanism for eliciting the *missing* information from the user [3:9].

[8]These "views" are also called models. The former is preferred here to avoid confusion with the *system* models already mentioned.

the target system. Correctness-preserving formal transforms are the mechanism used to derive the design specification from the formal (requirements) specification.

In the previous *AFIT*tool version of AWSOME, the models were represented and manipulated using Reasoning Systems' Refine language and AST manipulation functions. Recognizing an opportunity to improve the way *AFIT*tool represented object-oriented and imperative designs in the design model, Graham designed the wide-spectrum Common Object-oriented Imperative Language (COIL) [25]. Cornn followed by recognizing several commonalities between the analysis and design models and extended the language in a way that it could be used to model both. The KBSE research group then rebuilt *AFIT*tool around AWSOME using the Java™ programming language. The result is a formal, object-oriented transformation environment built around a wide-spectrum modeling language and the latest Java™ technology.

Figure 13 shows a subset of the model resulting from the revision of *AFIT*tool. Every entity inherits from `WsObject`. Since this methodology primarily uses `WsPackage`, `WsClass`, and `WsAssociation`, those entities are highlighted.

The surface syntax of AWSOME was still under revision at the time of this writing. The tentative surface syntax for the four primary constructs used in this research (i.e., the package, class, aggregation, and association constructs) is shown below.

- Type examples:

```
Type ZeroOrOnce    is   range 0..1;
Type ExactlyOnce   is   range 1..1;
Type ZeroOrMany    is   range 0..*;
Type OnceOrMany    is   range 1..*;
Type MyColor       is   (Red, Green, Blue);
```

- Package example:

```
Package MyArchitecture is
    ... <package declarations>
end Package;
```
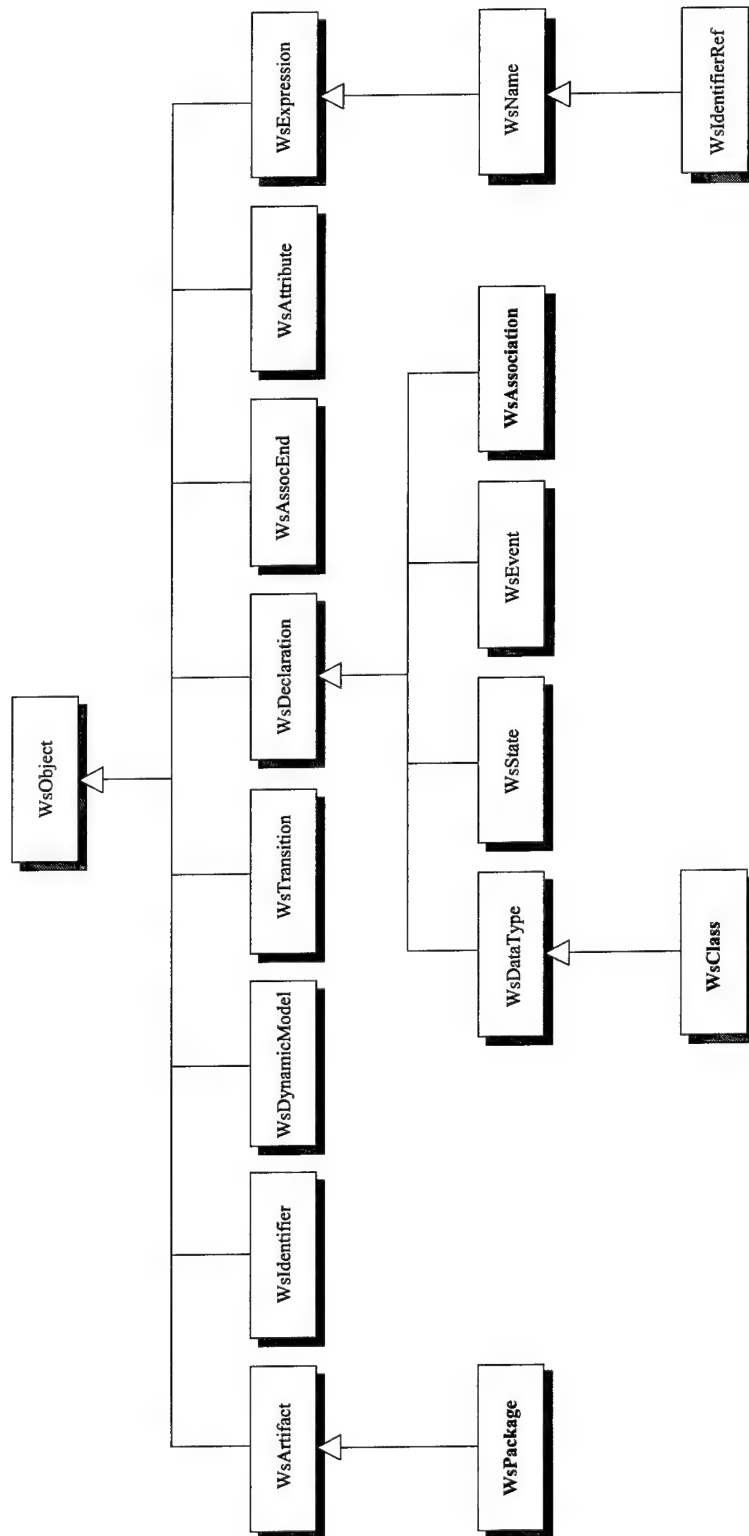
Figure 13.   Key elements of the AWSOME model.

- Class example:

```
Class MyInterface is
    var attribute1  : classX;
    var attribute2  : typeA;
    procedure InitInterface()
        guarantees (size(attribute2') = 0);
    invariant (attribute2.editable = True);
    dynamic model{
        state Start;
        state Ready;
        event DoIt;
        transition{
            initState      Start;
            receiveEvent   DoIt;
            nextState      Ready;
        }
    }
end Class;
```

- Aggregation example:

```
Aggregation MyInterface_has_port_RcvPort_DoIt is
    parent theComponent   : MyInterface      occurs ExactlyOnce;
    child  thePort        : RcvPort_DoIt     occurs ZeroOrOnce;
end Aggregation;
```

- Association example:

```
Association SndPort_attaches_to_InRole is
    role thePort          : SndPort          occurs ExactlyOnce;
    role theRole          : InRole           occurs ExactlyOnce;
end Association;
```

2.5 Summary and Conclusion

Software system architectures are extremely valuable abstractions in the software development process. The foundation has been laid for implementing such architectures in a formal, semi-automated fashion; but there is still much to be done. Shaw and Garlan established a conceptual framework for thinking about different styles of architecture. Gamma, et al, paved the way for cataloging reusable design patterns, but focused primarily on a level below that of software system architectures. Coplien and the PLoP community continue to address software design pattern reuse at all levels, although their approach is relatively informal and targets the expert architect performing manual system design rather than the semiautomated transformation system.

III. Modeling Non-hierarchically Homogeneous Architectures

3.1 Introduction

Every software system has an architecture, whether or not the architecture was intentionally designed during the development process. Moreover, most object-oriented, formal, transformational development environments will produce software systems that conform to the object-oriented architectural style. However, attempts to *explicitly* model an architecture during the transformation process—for the purpose of leveraging style-specific characteristics during design analysis, facilitating the low-level design transformation itself, or ensuring the conceptual integrity of the system during maintenance—are rare, indeed. To facilitate this modeling, a transformational environment must provide a sufficient representation mechanism—a modeling language capable of capturing the essential aspects of an architectural design.

This chapter introduces a broad classification of software architectures, discusses the different languages available for architectural specification, and presents a modified process model for explicitly modeling a simple class of architectures in an object-oriented, formal transformation system. The purpose of the NM-class transformation process is simply to lay the groundwork for a more robust set of transforms that will handle both NM-class and HH-class architectures. Without this follow-through, the NM-class architecture transformation process buys little more than design model overhead. Chapter IV defines the transforms required to model the broader class of architectures, and Chapter V provides a walk-through demonstration of both transformation processes.

3.2 Classes of Architecture

Classification of software architectures is an active area of research in the architectural community. Typically, the approach is to classify architectures based on the style they exhibit and to group these styles into families (see Table 7). Often, however, software architectures do not conform to singular styles. Bass, et al., label such architectures as *heterogeneous*, meaning they exhibit more than one style of architecture by incorporating—into a single system—properties from different styles [5:102]. Their *hierarchically hetero-*
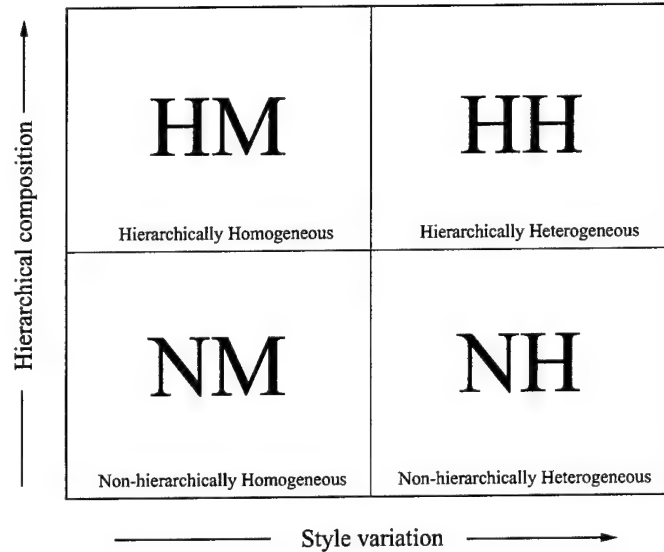
Figure 14.   Architectural dimensions.

geneous class of architectures refers to the way components of one style can consist of an arrangement of components and connectors (i.e., a sub-architecture) of yet another style. Their recognition of hierarchically heterogeneous architectures suggests a new and useful taxonomy based on two orthogonal dimensions—hierarchy and heterogeneity.

The *hierarchy* dimension addresses the degree to which components in the architecture are *hierarchical* or *non-hierarchical*. Hierarchical components, by definition, are composed of other components. Primitive components are those components that contain no others in their definition. Architectures that contain at least one hierarchical component are considered hierarchical architectures; those that do not are considered non-hierarchical architectures. This property of a class of architectures should not be confused with the *layered style* of architecture. A layered architecture (one that conforms to the layered style) is one that assigns components "to layers to control intercomponent interaction." This has little to do with nesting a configuration of one style inside a component of another style. A more detailed explanation of the layered style is provided by Bass, et al., and should clarify the difference [5:100–101].

The *heterogeneity* dimension focuses on the variety of styles used to architect the system. The styles manifest themselves as style properties such as component and connector

40

types, topological constraints, and semantic interpretation. Those architectures exhibiting properties defined by a single style are called *homogeneous* architectures, meaning they are a pure style; those that adopt properties defined by multiple styles are called *heterogeneous* architectures.

At the same time an architecture is non-hierarchical or hierarchical, it is also homogeneous or heterogeneous (see Figure 14). Therefore, a high-level taxonomy of architectures can be described. This taxonomy contains four classes of architecture defined in the following way[1]:

Non-hierarchically Homogeneous (NM): these architectures are *flat*—they contain no nested components. In addition, an NM-class architecture exhibits the properties a single architectural style. A good example is a pure and simple object-oriented system design without aggregate classes as shown in Figure 15. The components of the system are objects and the connectors are method calls between objects. There are no nested components defined by the system.

Non-hierarchically Heterogeneous (NH): these architectures are also flat, with no nested components. An NH-class architecture, however, displays properties defined by two or more architectural styles. There is a constraint, however, that only components sharing a common connector type can be connected. In other words, the only way to connect non-hierarchical components typed by two different styles of architecture is through a commonly defined connector. Theoretically, two different styles *could* share connector and component types and be differentiable only by their semantic and/or topological constraints. In such a case, the two styles could be interleaved to form an NH architecture.

Hierarchically Homogeneous (HM): these architectures are hierarchical, containing nested components. Like the NM-class of architectures, an architecture in this class is constrained to the properties of a single style. A good example is an object-

---

[1]The figures depicting these classes of architecture utilize a notation invented by the author for the purpose of explaining the classification scheme. The component and connector shapes are style-neutral and are only meant to convey the existence or non-existence of heterogeneity and hierarchy.
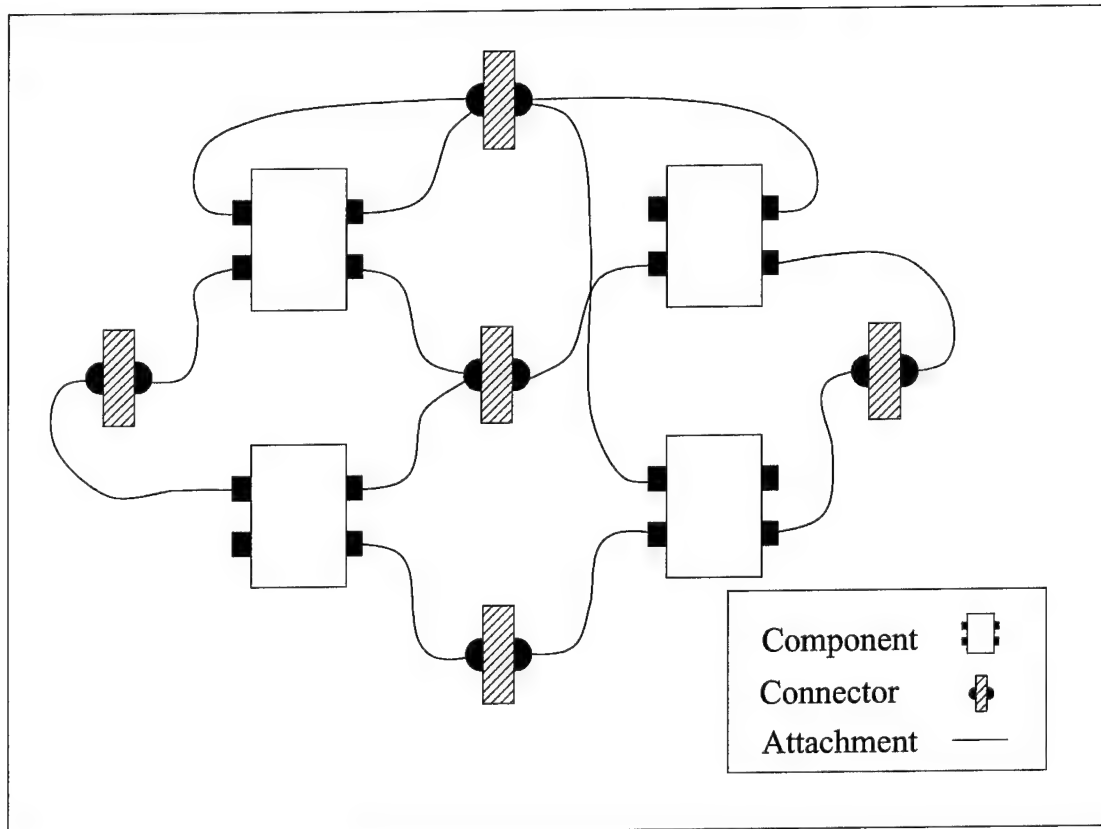
Figure 15. A non-hierarchically homogeneous (NM) architecture.
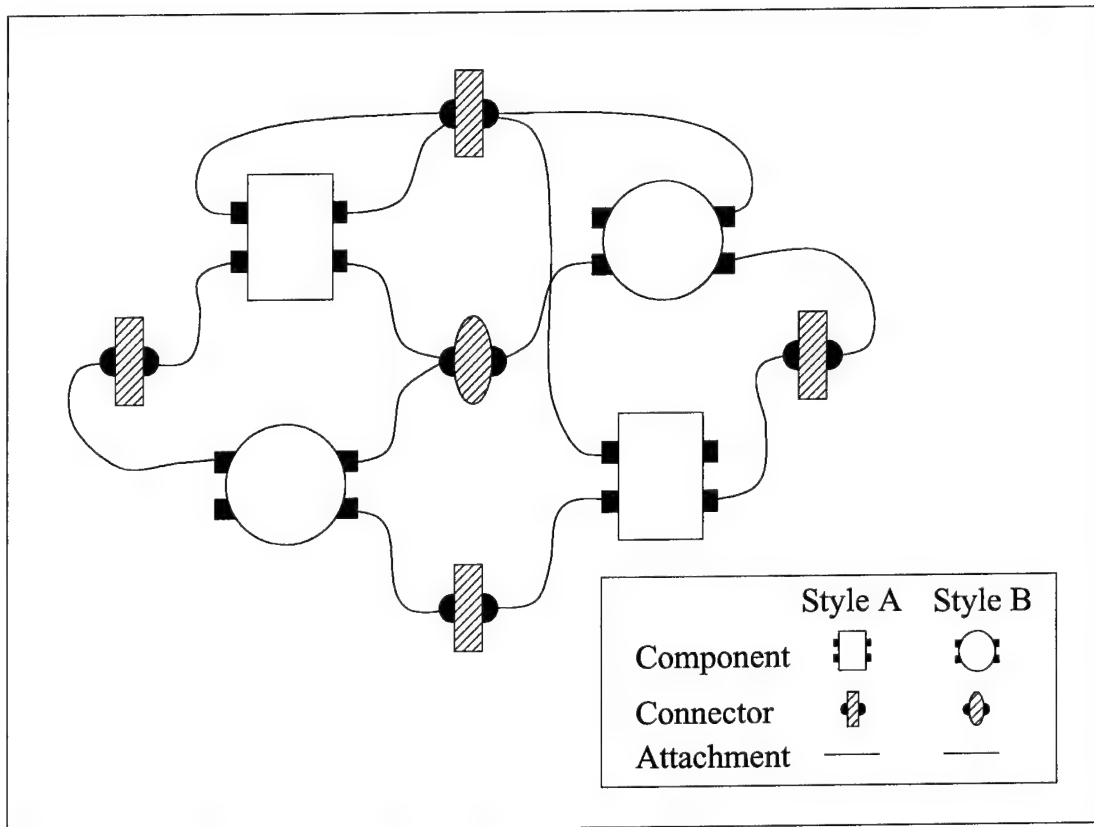
Figure 16.   A non-hierarchically heterogeneous (NH) architecture.
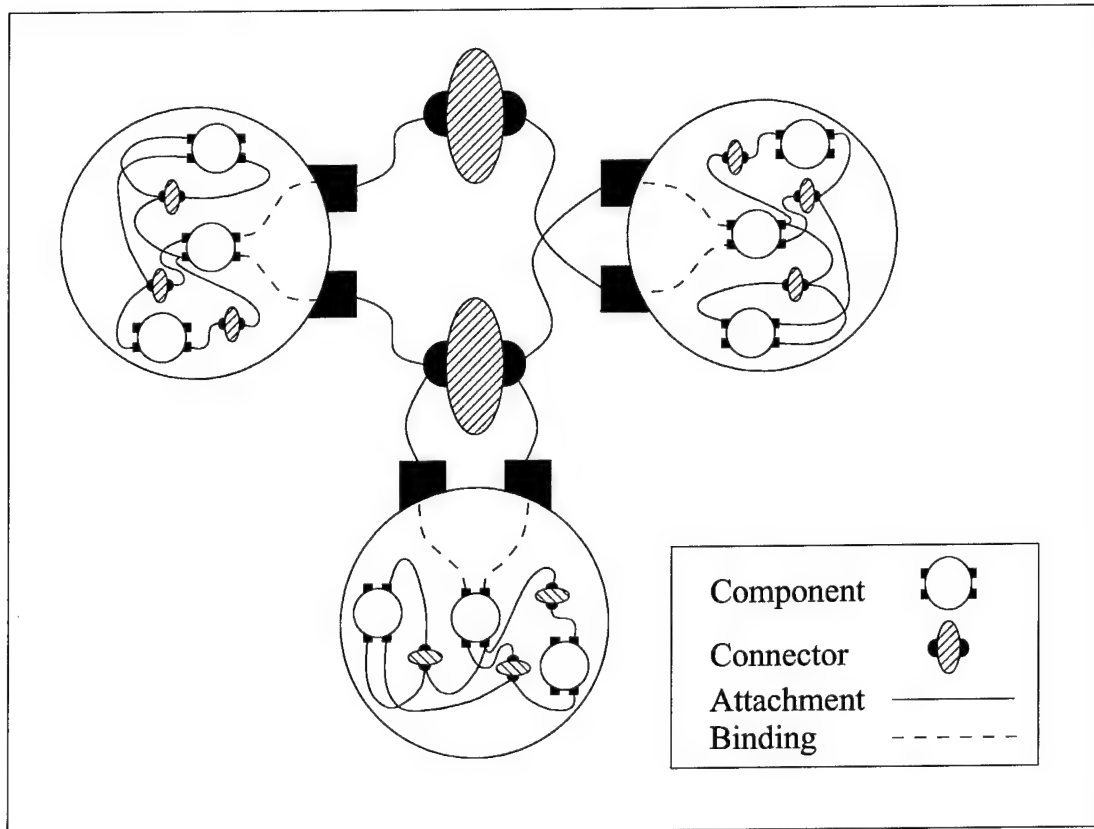
Figure 17. A hierarchically homogeneous (HM) architecture.

oriented system design with aggregate objects modeled as architectural components (see Figure 17).

Hierarchically Heterogeneous (HH): these architectures are hierarchical, containing nested components. In other words, a component at one level in the hierarchy is represented internally as a configuration of components (i.e., the lower level). Additionally, an architecture in this class exhibits properties of two or more styles usually, but not necessarily, at different levels in the hierarchy. An example of an HH-class architecture is an object-oriented client-server system (see Figure 18). In this architecture, the system level components are *clients* and *servers*, and the system level connectors are remote procedure calls (RPC). Each client and server component is an *aggregate component*, i.e., is composed of a configuration of one or more sub-components. Connectors between the sub-components in this example are method
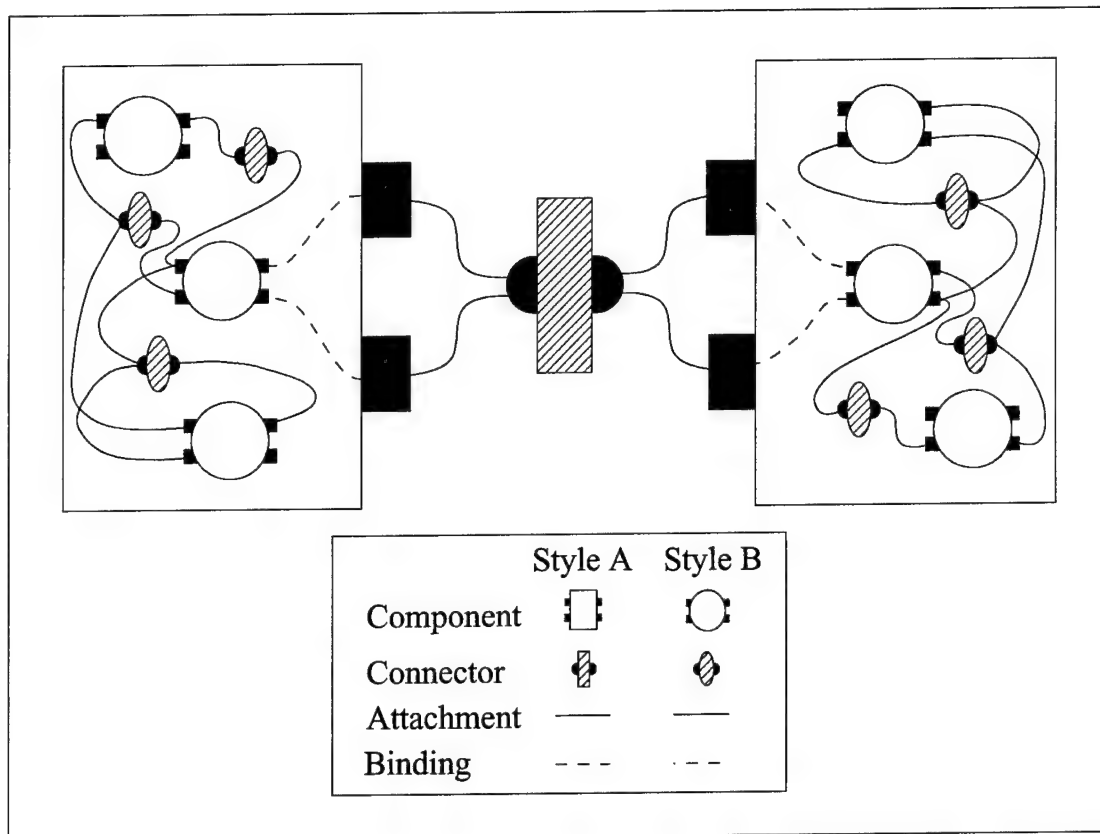
Figure 18. A hierarchically heterogeneous architecture.

calls, while connectors between the client and server components themselves are RPCs.

AWSOME, and similar object-oriented transformation systems, are generally designed to model only homogeneous architectures—both non-hierarchical (NM) and hierarchical (HM)—based on the object-oriented architectural style. This is because all components and connectors are assumed to be objects and method calls, respectively. Getting a transformation system to accurately manipulate homogeneous configurations as has been done in the AWSOME environment is progress, but even more can be done to enable semi-automated production of complex real-world software systems by integrating a rich architectural modeling capability. This capability begins with a modeling language.

45

3.3   Choosing an Architectural Description Language

There are many languages available for the description of software architectures—some designed for the purpose and others adapted to it. The languages designed explicitly for the representation of software architectures are the class of languages called architectural description languages (ADL). Several ADLs have attracted attention in the research community—Aesop, Wright, and Rapide are commonly referenced in the literature. These languages were designed for use in specific environments targeting goals germane to those environments. They each have strengths, but are largely tied to the environments for which they were designed.

Alternatively, one could develop a new, tailor-made ADL that meets the needs of a specific development environment. Using the Acme interchange language as a guide, one could create an ADL rich enough to allow site-unique modeling and analysis while, at the same time, ensuring the ability to export the model for analysis using third-party tools. While such an approach might directly address the unique needs of the developing organization, it would require a great deal of cost and effort, both in developing and maintaining the language and in training personnel to use it, and would likely increase the complexity of the environment by adding yet another dialect to the mix.

The final approach, one that is especially viable in a formal, object-oriented, transformational environment, is to model the architecture in an object-oriented fashion, using the language already being used to represent analysis and design specifications. The languages with which most are familiar (e.g., Java, C++, Ada95) are geared toward representation of implementation level entities like classes, types, variables, procedures, and functions. Languages that are used to model various levels of abstraction (i.e., analysis, design, and implementation) are called *wide-spectrum* languages. Wide-spectrum languages must account for the more abstract notions found at the analysis and design levels like inheritance, aggregation, associations, and the dynamic behavior of the system being developed. The architecture, falling between analysis and design in both the abstract modeling sense and in the development process itself, introduces new entities such as architectural *components*, *connectors* between components, and the *attachments* between component *ports* and connector *roles* (see Section 2.2.1 for a description of these entities).

Initially, the objective was to propose an expansion of the wide-spectrum language to include these entities as first-class constructs in the language. Such an approach would appropriately emphasize the importance of the architectural abstraction and provide a robust model for the task. However, it quickly became evident that the architectural entities can be modeled using the object-oriented wide-spectrum language as-is, thereby avoiding the need to increase the complexity of the language and the modeling environment with new constructs and operations to manipulate them. After all, a component or connector is simply another *class* of objects in the real world (from the perspective of the software architect). In an environment already geared toward transforming object-oriented analysis models to object-oriented design models using a wide-spectrum language, adopting the existing language as-is to represent architectures seemed the best approach.

3.4   Essential Elements of Architecture Models

Knowing what surface and abstract syntax to use for architectural modeling is only half the battle; modeling the *right* architectural elements with appropriate language constructs is key to the success of any such endeavor. It is important to recall the primary reasons for modeling the architecture of a software system.

First, the architectural transformation from the analysis model provides a mechanism for asserting high-level design decisions that will enrich the low-level design transformation. One way it does this is by eliminating from the solution space all solutions that do not conform to the specified architectural style. For instance, transforming an object-oriented analysis model directly to an object-oriented design with no entry point for decisions about how to organize the classes in the system nets a pure object-oriented architecture. Architectural decisions do not replace low-level design transforms; rather, they enrich the process by allowing the software engineer to *constrain* the subsequent low-level design transforms and produce, perhaps, a more robust solution to the customer's problem.

Second, the architectural design represents the earliest point at which the developing organization can evaluate the efficacy of a particular solution. Requirements analysis focuses, for the most part, on modeling the problem specification with as little thought as possible given to specific solutions or implementations. Once the problem specification

47

is complete, however, the developers transition to solution-oriented thinking. The architecture represents the most abstract model of a solution to the customer's problem. It can be evaluated, before any further work is performed, for its ability to satisfy critical requirements and quality attributes demanded by the customer. This analysis can be accomplished either by using native analysis tools or by piping the architecture through an interchange language to a more analytically disposed development environment. If found inadequate, the model can be modified or discarded altogether as a feasible alternative. By explicitly modeling the architecture and introducing an analysis capability built over that model, one can improve the likelihood of customer satisfaction before generating the low-level design and implementation.

Finally, the architectural model stands as a guard-post against changes to the system that may result in a deterioration of its conceptual integrity. It is well understood that software systems deteriorate and grow increasingly complex over time as new customers demand a variety of enhancements, and developers, many quite unfamiliar with the original intent of the software systems they maintain, attempt to incorporate fixes and enhancements. An architectural abstraction shouts the "big picture" to all who would attempt to insert an *innocent* modification and, thereby, reduces the likelihood of conceptual deterioration.

To do these things, the architecture should reflect an abstraction of the target system that excludes the internal design of components. It must focus on the assignment of responsibilities to components and connections between those components, abstracting all else. The remainder of this chapter explores a method for explicitly modeling the non-hierarchically homogeneous (NM) class of architectures and paves the way for the more interesting class of hierarchically heterogeneous systems. A prescription for the *right* architectural elements and the language constructs used to model them is presented in the following subsection.

3.5 Essential Elements of Non-hierarchically Homogeneous Architectures

For the purposes of this thesis, when referring to an NM-class architecture, it is assumed that the homogeneous style in use is the *object-oriented* style. This assumption

arises from the fact that this thesis targets formal object-oriented environments and the object-oriented systems they produce[2]. Also, note that little is gained in an object-oriented environment by explicitly modeling the object-oriented architecture than an increase in the size of the model. In an environment capable of producing only non-hierarchical and hierarchical object-oriented architectures, the architecture can be *inferred* from the design model or, for that matter, from the analysis model and this explicit modeling of the architecture is unnecessary overhead. This discussion of the requirements for modeling an NM-class of architectures, however, lays the foundation for an environment that can also model heterogeneous architectures where the architectural styles employed can no longer be easily inferred. Such an environment is discussed in the next chapter.

To effectively capture the essence of an NM-class architecture, the modeling language must be able to represent the following elements: the system itself (i.e., the encapsulating entity), primitive components[3], ports, connectors, roles, and attachments. Figure 19 graphically depicts a simple object-oriented NM architecture and is included for reference. The architectural model must be uniquely identifiable in the abstract representation of a complete system model (i.e., one that includes analysis and design models). The root node of the architecture model must be represented using a named entity that can contain other entities. This parallels the use of *packages* by some vendors in the object-oriented arena, where a package can contain other entities such as classes. Others provide a similar notion, while giving it a different name. The term "package" will be used for the purposes of illustrating the requirement to represent the root node of an architectural model.

Components, ports, connectors, and roles are all modeled using the *class* construct. A class is a named entity that contains a reference to its superclass, a set of attributes, a set of operations, and a dynamic model. Using a class to model the components and connectors in the system adds greatly to the model. The architect can *type* the components in the architecture by using class inheritance, allowing subsequent type-checking of the

---

[2]To be sure, an object-oriented transformation system can produce architectures exhibiting different styles, but they are inherently hierarchically heterogeneous since the lowest level components and connectors are assumed to always be objects and method calls, respectively.

[3]The term "primitive component" is used to distinguish components of non-hierarchical architectures from the aggregate components found in hierarchical architectures.
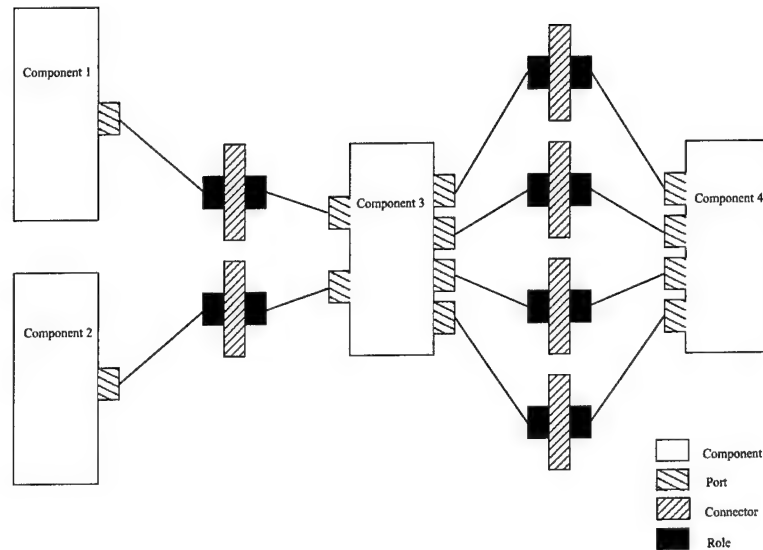
Figure 19.   NM Object-Oriented Architecture.

Architectural Element	Language Construct
system architecture	package
component	class
port	class
connector	class
role	class
attachment	association

Table 4.   NM architectural element to language construct mappings.

architecture during analysis. In addition, the architect can assign ports to components and roles to connectors by using the aggregation mechanism provided by the language.

The attachments between ports and roles are modeled using the *association* construct. An association is normally used in object-oriented analysis modeling to associate classes in the specification. Since the association is used to associate classes, and the ports and connectors are modeled using classes, the association can be used to represent architectural attachments. A graphical representation of an attachment is shown in Figure 20.

This approach to capturing the essence of NM architectures capitalizes on the availability of a wide-spectrum object-oriented language by using the preexisting constructs in a way that simplifies the insertion of an architectural abstraction level and precludes the need to add new constructs to the language (see summary in Table 4). Figure 21 depicts
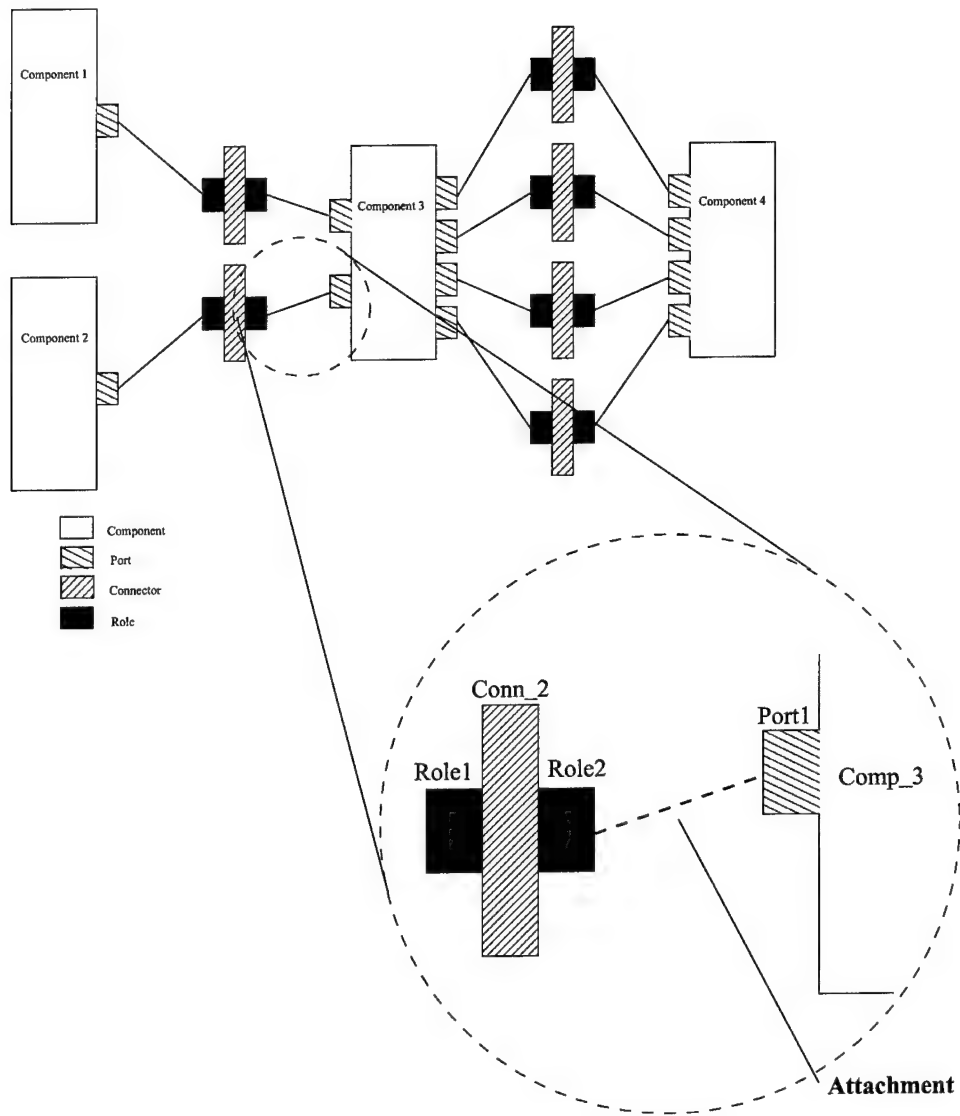
Figure 20.   An example architectural attachment.

```
package OO_Architecture is

    Class Comp_1 is Component with
    end Class;

    Class SndPort_A is Port with
    end Class;

    Aggregation Comp_1_has_port_SndPort_A is
        parent theComponent      : Comp_1      occurs ExactlyOnce;
        child  thePort           : SndPort_A   occurs ZeroOrOnce;
    end Aggregation;
    .
    .
    .
    Class InRole is Role with
    end Class;

    Class Conn_A is Connector with
    end Class;

    Aggregation Conn_A_has_role_InRole is
        parent theConnector      : Conn_A      occurs ExactlyOnce;
        child  theRole           : InRole      occurs ZeroOrOnce;
    end Association;
    .
    .
    .
    Association Comp_1.SndPort_A_attaches_to_Conn_A.Inrole is
        role thePort        : Comp_1.SndPort_A  occurs ExactlyOnce;
        role theRole        : Conn_A.InRole     occurs ExactlyOnce;
    end Association;
    .
    .
    .
end Package;
```

Figure 21.   Simple object-oriented architecture.

simple architecture shown in Figure 19 using the AWSOME language.

3.6   Modeling Non-hierarchically Homogeneous Architectures

The discussion in Section 3.5 established the elements necessary to model a non-hierarchically homogeneous architecture. From that foundation, the specific modeling technique can be established. In other words, the specific transformations necessary to derive an architectural model from the analysis model are presented. Note that these transformations are automatic. Each style of architecture will have different rules applied during the transformation process. With the exception of the object-oriented architectural style, the definition of these rules is outside the scope of this thesis.

For an object-oriented style, there are few rules to be applied. First, there is a one-to-one correspondence between a class in the analysis model and a *component* in the architecture, multiplicities notwithstanding. Since objects are often generated dynamically during runtime as a result of many non-deterministic input sequences and cannot, therefore, be anticipated during the development of the system, this methodology simply models a single component for a single class. Second—based on the author's assumption that an event in an object-oriented analysis model translates to a method call in the design—there is a one-to-one correspondence between events in the analysis model and *connectors* in the architecture. Third, since an object-oriented event is a communication mechanism between two classes, there are two *roles* created for each connector—one for the *caller*, the other for the *callee*[4]. Fourth, for each class in the analysis model and for each event sent or received by the class, a *port* is created for the corresponding component in the architecture. So, if a class responds to three events and sends two events, its corresponding component will have a total of five ports. The following subsections present the transformations necessary to handle each element category.

3.6.1 Transform NM_0: The Architectural Model. The first transformation[5] is the simplest one; it establishes the root node of the AST that will eventually house the architectural model as a child of the existing system level AST and creates nodes to represent the abstract classes in the architecture (e.g., Component, Connector, etc.). These classes are used to *type* the architectural entities. In this way, a class representing a component can be distinguished from a class representing a connector not only by the prefix given to the name, but also by its defined superclass. The root of the existing system model is accepted as input to the transformation. For the purposes of this illustration, the root node of the AST containing the entire system model will be referred to as `SysModel`. The NM_0 transform is defined in the following way:

---

[4]Note that this assumption—two roles per connector—may not hold for non-object-oriented architectures.

[5]Note that the transformation numbering is zero-based. This method was chosen because the initial transformation is not really so much a transformation as it is a preparatory function.
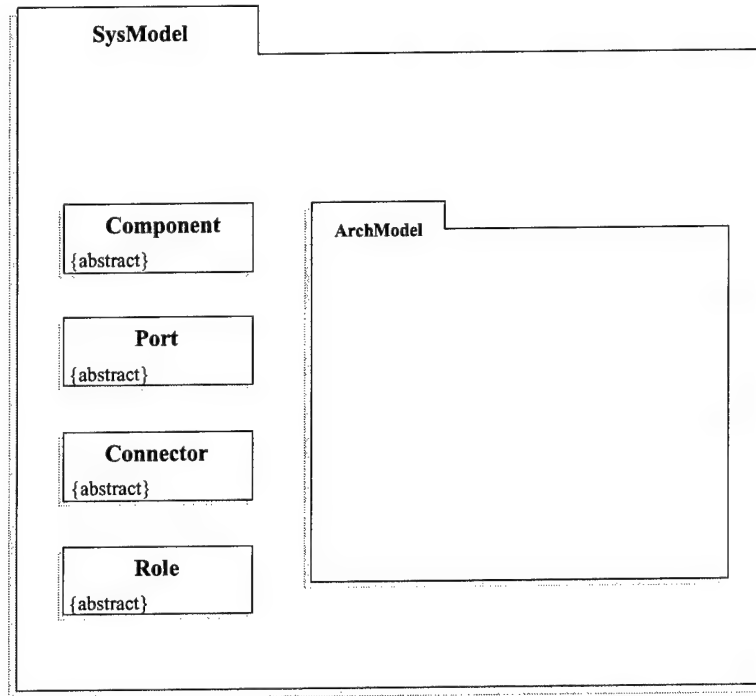
Figure 22. Architecture model after Transform NM_0.

Transform NM_0:

1. *Instantiate one package node in the model as a child of* `SysModel`. *Name the new package node* `ArchModel`.

2. *Instantiate one abstract class node in the model, as a child of* `ArchModel`, *for each of the following elements:* `Component`, `Connector`, `Port`, *and* `Role`.

3. *Return* `SysModel`.

The results of this transform are graphically portrayed by Figure 22.

3.6.2 Transform NM_1: The Components. This transform accepts `SysModel` as input and is defined in the following way:

Transform NM_1:

1. *For each original class in the analysis model, instantiate one class node as a child of* `ArchModel`, *naming each new node* `Comp_classname`, *where*

54
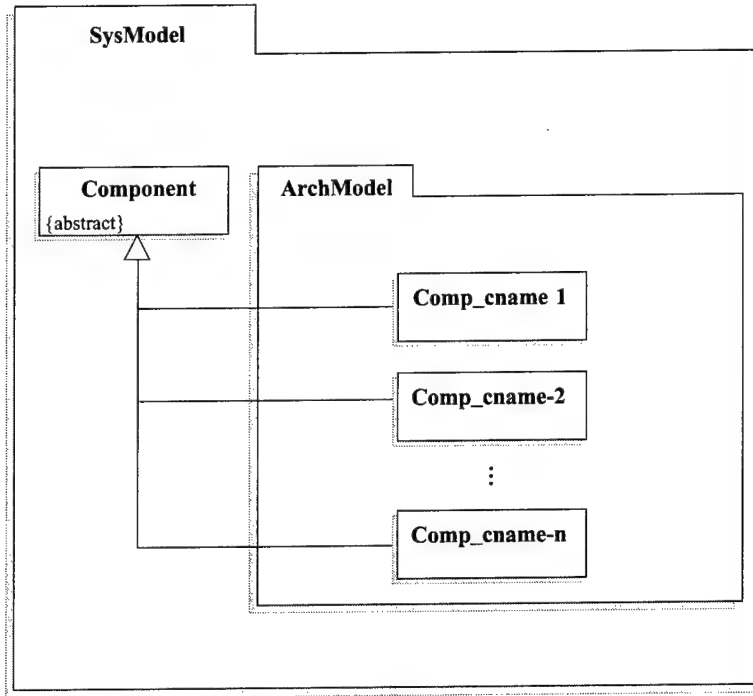
Figure 23.   Architecture model after Transform NM₁.

classname is the name of the originating class. Set the `Component` *class as superclass for* `Comp_classname`.

2. *Return* `SysModel`.

The results of this transform are graphically portrayed by Figure 23.

3.6.3   Transform NM₂: The Component Ports.   This transform accepts `SysModel` as input and is defined in the following way:

Transform NM₂:

1. *For each original class in the analysis model and for each event received or sent by the class, instantiate one class node as a child of* `ArchModel` *to represent the port. Also, make the new port class an element of the aggregate class that models the associated architectural component, using the aggregation language construct. Name each new port node* `direction`Port_eventname, *where eventname is the name of the originating event, and direction is "Snd" or "Rcv," depending on whether*

55

the event is a send or receive event. Set the `Port` *class as superclass for*
direction `Port_eventname`.

2. *Return* `SysModel`.

The results of this transform are graphically portrayed by Figure 24.

3.6.4 Transform NM₃: The Connectors.   This transform accepts `SysModel` as
input and is defined in the following way:

Transform NM₃:

1. *For each original event in the analysis model, instantiate one class as a con-*
   nector node in `ArchModel`. *Name each new connector node* `Conn_eventname`,
   where eventname is the name of the originating event. Set the `Connector`
   class as superclass for `Conn_eventname`.

2. *Return* `SysModel`.

The results of this transform are graphically portrayed by Figure 25.

3.6.5 Transform NM₄: The Connector Roles.   This transform accepts `SysModel`
as input and is defined in the following way:

Transform NM₄:

1. *For each original event in the analysis model, instantiate two class nodes*
   as children of `ArchModel`. *Also, make the new role nodes elements of the*
   aggregate class that models the associated architectural connector. Name
   one role node InRole_`eventname` *and one role node* OutRole_`eventname`,
   where eventname is the name of the originating event. Set the `Role` *class*
   as superclass for InRole_`eventname` *and* OutRole_`eventname`.

2. *Return* `SysModel`.

The results of this transform are graphically portrayed by Figure 26.
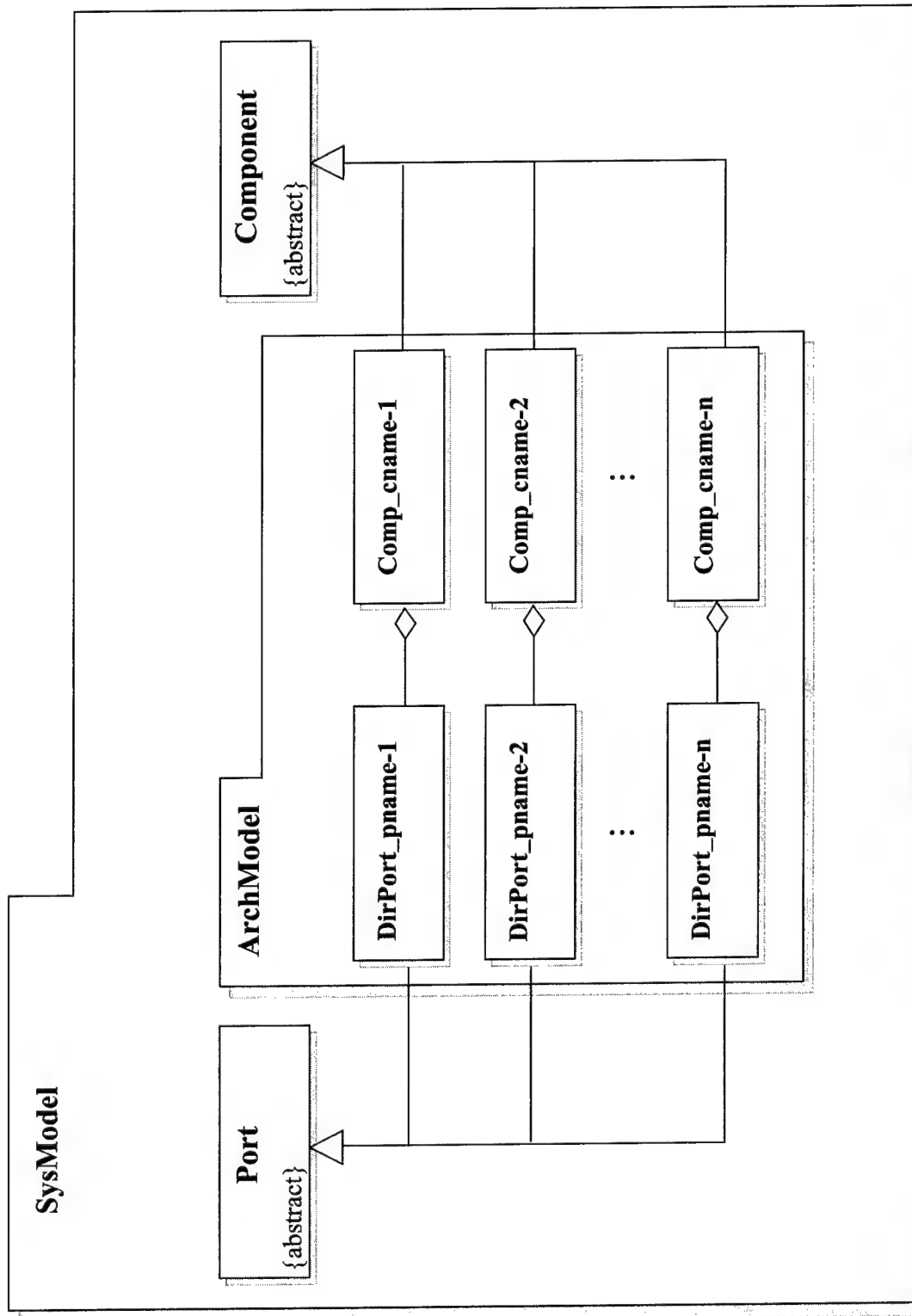
Figure 24. Architecture model after Transform NM$_2$.

57
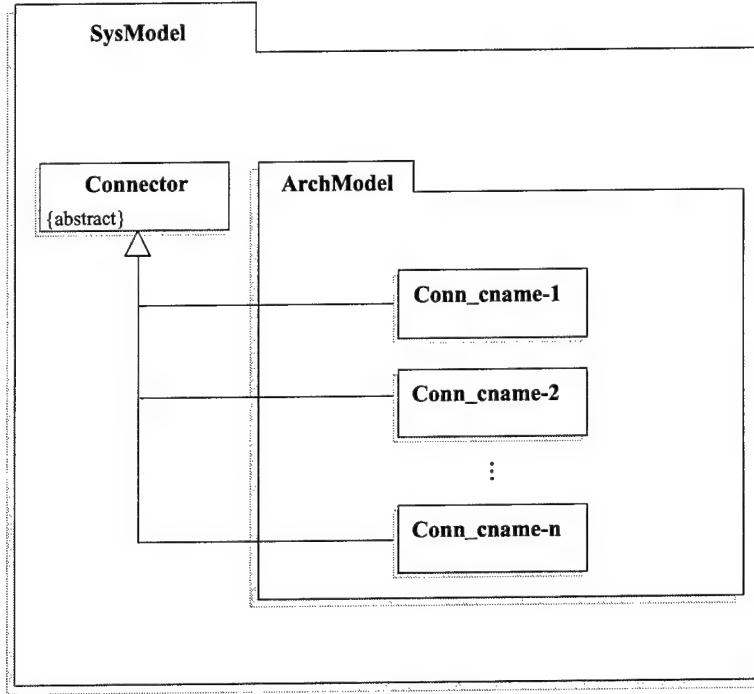
Figure 25. Architecture model after Transform NM_3.

3.6.6 Transform NM_5: The Port-to-Role Attachments. This final transform is
responsible for creating the nodes in the AST that represent attachments in the architec-
ture. Architectural attachments associate component ports to their respective connector
roles. These attachments constitute the thread that binds the nodes in the architectural
AST into a meaningful architectural model. The transform accepts SysModel as input and
is defined in the following way:

Transform NM_5:

1. *For each original class in the analysis model and for each event sent
   or received by the class, instantiate one association as a child node of
   ArchModel to represent the attachment. The association node will asso-
   ciate the port node—defined for the event sent or received by the orig-
   inal class—with the appropriate role node in the following way: if the
   event was received by the original class in the analysis model, the as-
   sociation links the port to the _Out role of the connector; if the event
   was sent by the original class in the analysis model, the association links*

58
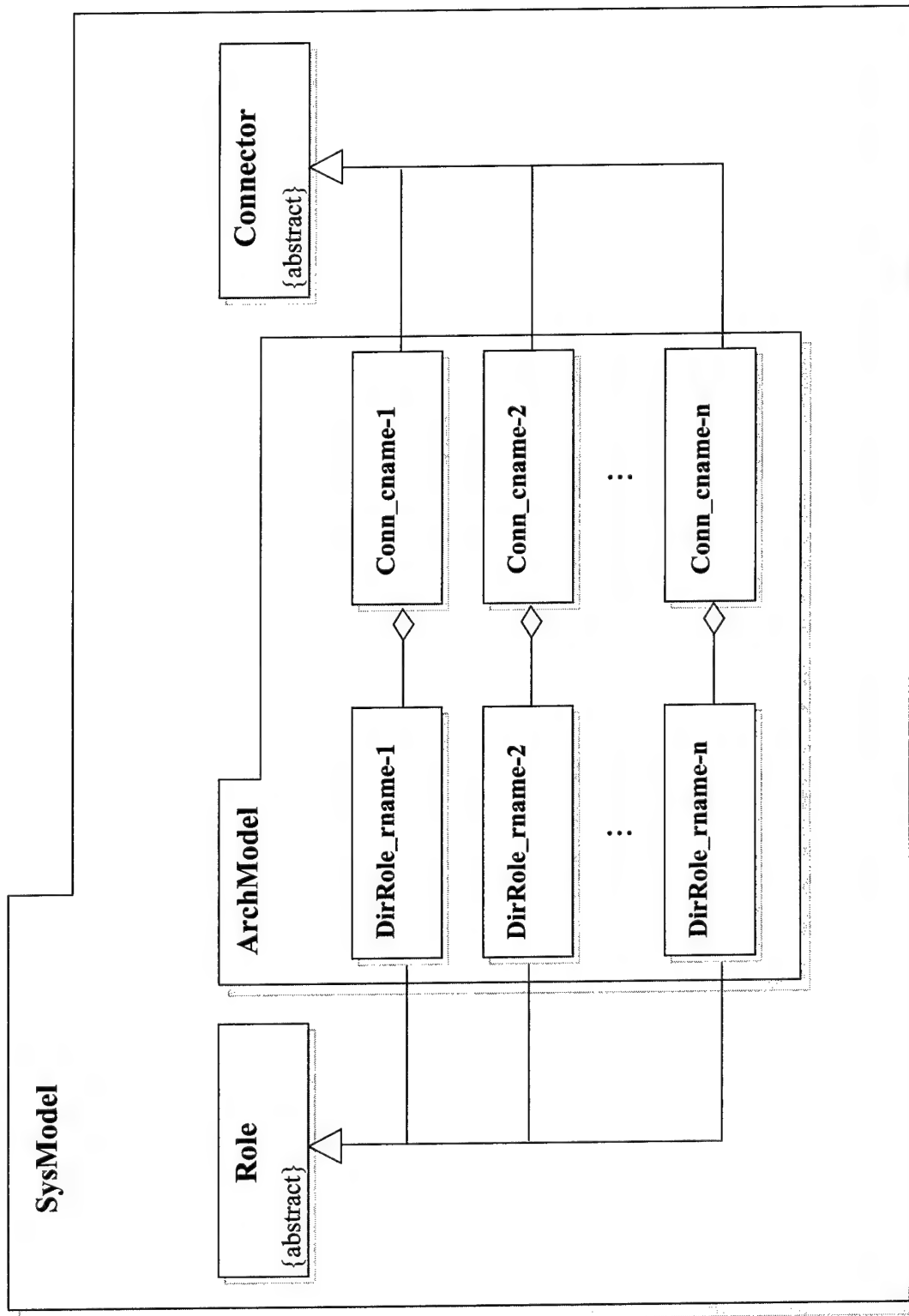
Figure 26.   Architecture model after Transform NM$_4$.

59

the port to the _In role of the connector. Name each attachment node `dirPort_attaches_to_dirRole` *where* `dirPort` *and* `dirRole` *are the names of the port and role involved in the attachment.*

2. *Return* `SysModel`.

The results of this transform are graphically portrayed by Figure 27.

3.7 Summary

Software architectures can be broadly classified as: non-hierarchically homogeneous, non-hierarchically heterogeneous, hierarchically homogeneous, or hierarchically heterogeneous. Most object-oriented, formal transformation systems operate by transforming an analysis model directly to a design model and conduct architectural modeling and reasoning functions only through implicit transformation and inference, respectively. In an environment supporting the production of only object-oriented architectures (hierarchical or non-hierarchical), the architectures can be explicitly modeled following the requirements specification phase and prior to low-level design transformation. However, in this limited environment, such an operation adds little more than modeling overhead. Having demonstrated the ability to explicitly model the architectures, however, one can construct an environment capable of modeling a wide variety of heterogeneous architectures. In this case, the architecture serves as an abstract constraint on the solution space and improves the ability of a transformation system to generate complex software systems.

60

Figure 27.   Architecture model after Transform NM$_5$.

61

IV. Modeling Hierarchically Heterogeneous Architectures

4.1 Introduction

The previous chapter presented a method for explicitly modeling non-hierarchically homogeneous (NM) architectures. The methodology proposed here builds on that foundation and provides a way to capture the class of architectures known as hierarchically heterogeneous (HH) architectures.

4.2 A Modified Transformational Process Model

Inferring object-oriented architectures from object-oriented analysis models and inserting the results into a design tree can be accomplished without changing the way a transformational system operates, as shown in the previous chapter. The notion of hierarchically heterogeneous architectural models, however, requires a change in the process model to allow for the introduction of new information during the transformation process. An architecture modeling tool, inserted between the problem setting phase and design phase, provides this capability.

It is important to note that the production of HH architectures requires additional human intervention during the transformation process. This intervention is required because the assumption that each class in the object-oriented analysis model transforms directly to a component in the architecural model is no longer valid. This methodology assumes several styles of architecture, and the component and connector types they offer, are available to the system architect. Figure 28 graphically portrays an environment supporting this approach (compare to Figure 12 in Chapter II).

Using such an environment, the architect first *profiles* the problem based on the software quality attributes required by the customer and then selects an appropriate composition of styles based on that profile[1]. The engineer graphically composes the architecture, using a graphical architecture modeling tool designed for the purpose, by assigning architectural constructs provided by the tool to entities in the analysis model, using whatever

---

[1]The actual profiling and style selection processes are outside the scope of this thesis and left for future research.
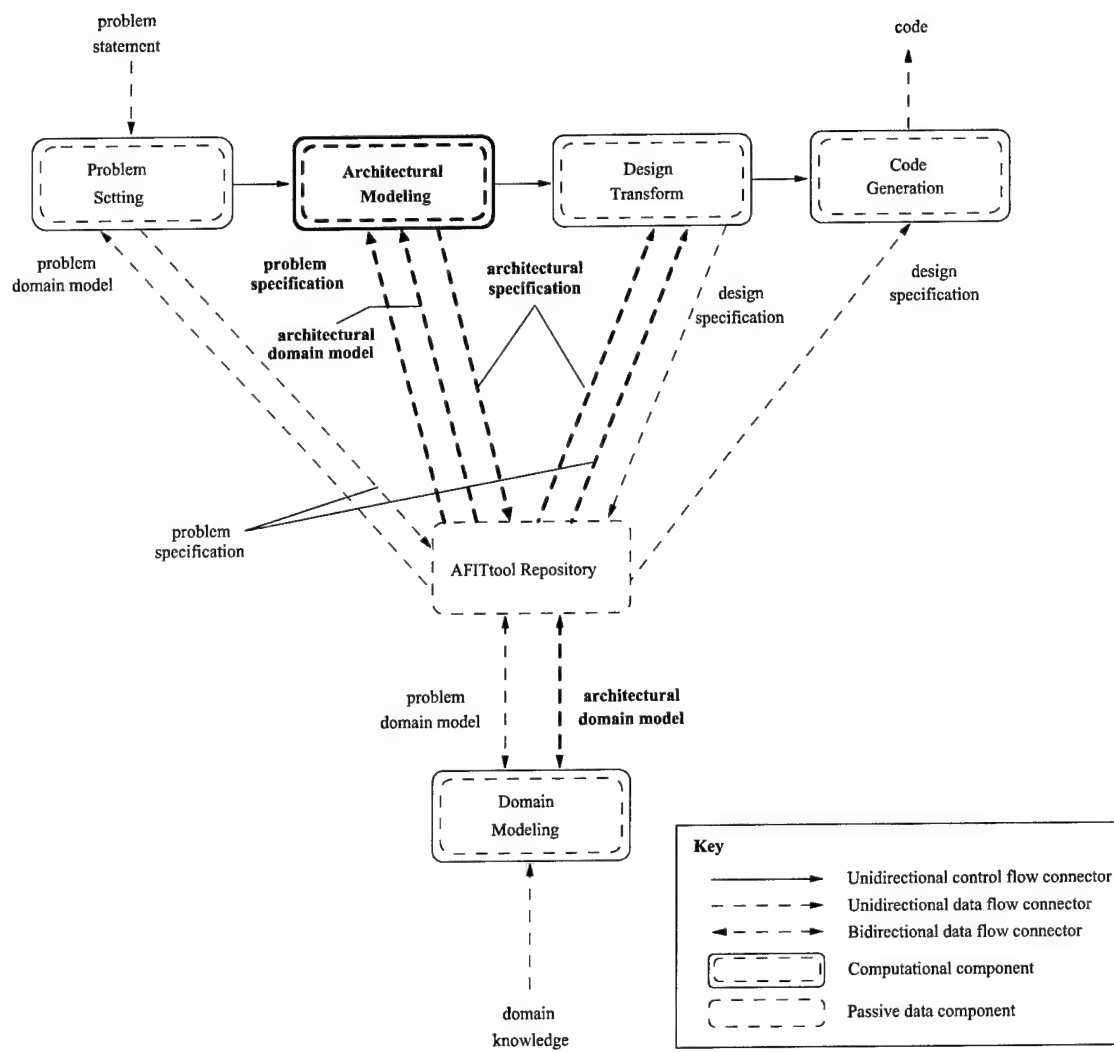
Figure 28.   An architecture-friendly process model.
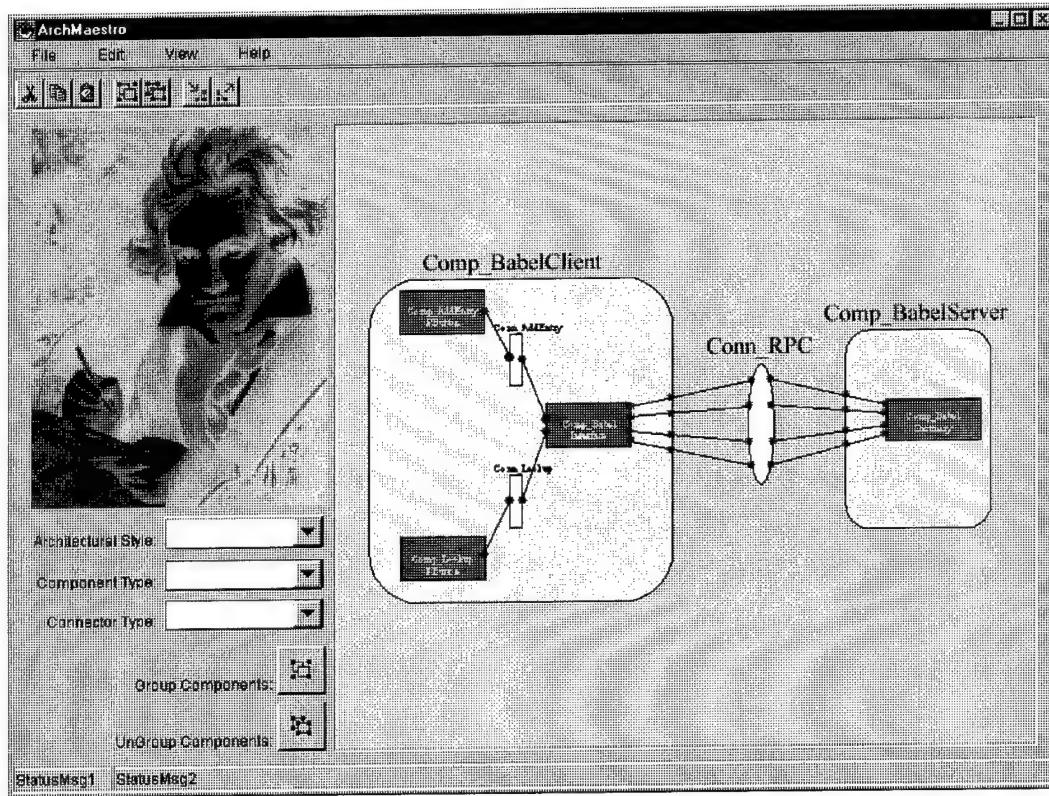
Figure 29.   An example architecture assistant GUI.

degree of hierarchy and heterogeneity is desired[2]. A screen-shot of a mock-up version of such a tool is shown in Figure 29. While the development of a tool to facilitate graphical architectural modeling is outside the scope of this research, such a tool is precisely what should be used to perform the transformations described. It would essentially take an analysis model as input, provide a way to overlay an architecture onto the model, and output the resulting high level design for use in further design transformations. Within this context, the necessary transformations from the analysis to an architectural model are presented.

---

[2]The tool would be designed to enforce semantic and topological constraints during the composition process, thereby ensuring the conformance of the architecture to published architectural styles.
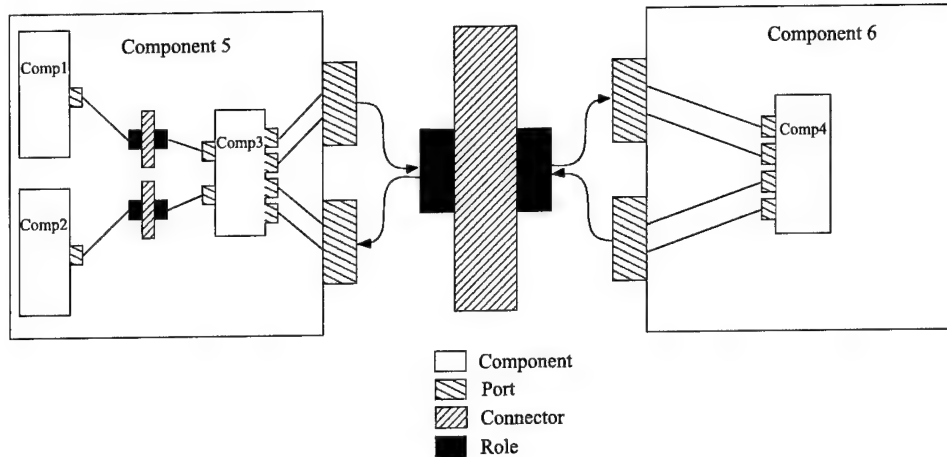
Figure 30. An object-oriented, hierarchically heterogeneous architecture.

4.3 Essential Elements of Hierarchically Heterogeneous Architectures

Like the NM-class of architectures presented in Chapter III, HH-class architectures necessitate modeling components, connectors, ports, roles, and attachments. In addition, HH-class architectures require representation of architectural styles, style-specific component and connector types, aggregate components, and port-to-port bindings. Figure 30 graphically depicts an object-oriented HH architecture and is included for reference throughout the following discussion.

To facilitate the use of various styles of architecture, the model must be able to represent a *style*. This can be done as before with the class construct, where the class is given a name identifying the style. Specific architectures can then be associated with the style. Likewise, the component and connector types are modeled as classes and are associated with their parent architectural styles. This modeling of the architectural style and component types is handled separately from the architecture of a specific system. The system architecture simply has an association with the style and the architecture's components and connectors inherit their properties from the types defined for the style.

The second change introduced by this class of architectures is the notion of nesting components, creating the hierarchy suggested by the class. In the simplest terms, just as an architecture contains components, connectors, ports, roles, and attachments, a component can be defined as consisting of a lower-level architecture of the same or of a different style.

65

Architectural Element	Language Construct
architectural style	class
component type	class
connector type	class
aggregate component	package
binding	association

Table 5.   HH architectural element-to-language construct mappings.

In other words, a component may encapsulate a sub-architecture. Therefore, this new aggregate component, like the system-level architecture itself, is modeled using the package construct. Primitive components are still modeled as classes. To capture the aggregate component's type, a reference (i.e., a class with the same name as the package representing the aggregate component) is created that refers back to the component type class. For example, to model an aggregate component called A that is an X_Type component, a package is created and named Comp_A. In addition, a class is created, named Comp_A, and is made a sub-class of X_Type. This overhead is one result of the decision to overload the existing wide-spectrum language for architectural specification.

The addition of an aggregate component introduces another issue that must be addressed in the model for HH-class architectures. In a flat architecture, components are linked together with connectors and the components and connectors are linked via port-to-role attachments. In such a case, everything is clearly connected. However, there must be a way to show how the sub-architecture modeled in an aggregate component communicates with the parent component. The construct commonly used to effect this connectivity is the *binding*. A binding links a port of a component in the sub-architecture to a port of the parent component. Figure 31 provides a graphical representation of a binding.

Therefore, the model must account for these port-to-port bindings. The most obvious solution to the problem is to use the association construct for the task. Since ports are classes and associations are used to link classes, using associations to model port bindings is a straightforward approach. In this way, the architecture will use the association language construct to model two architectural entities: port-to-role attachments and port-to-port bindings. Table 5 summarizes the use of object-oriented language constructs to model the architectural entities in HH-class architectures.

66
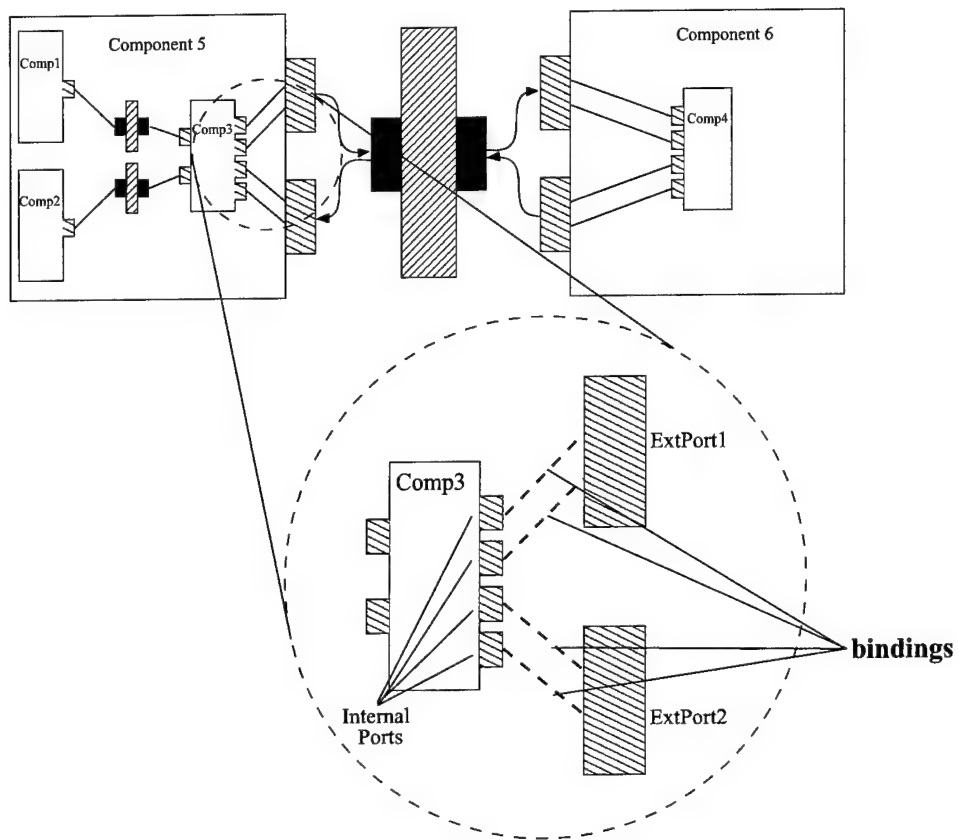
Figure 31.   An example binding in an HH-class architecture.

4.4 Modeling Hierarchically Heterogeneous Architectures

The previous discussion in Section 4.3 identified the elements necessary to model a hierarchically heterogeneous architecture. From that foundation, the specific AST modeling technique can be established. The transformations necessary to generate the architectural model are presented in the following subsections. The following subsections present the transformations necessary to handle each element category.

Recall that the transformations for NM-class architectures assumed an object-oriented environment producing systems that conform to the *object-oriented* architectural style. In such an environment, many assumptions can be made during the transformation about the architectural style and component and connector types (e.g., each component in the target architecture is generated from an object in the analysis model). In an environment designed to produce HH-class architectures, all bets are off—these assumptions can no longer be made. In addition, even though the elements of an NM-class architecture are present in an HH-class architecture, the transformations required to produce the latter will change due to structural changes in the target model.

There are a total of eight transforms used during the transformation process for HH-class architectures. The HH_0 transform sets up the model much like the NM_0 transform. Transforms HH_1 through HH_6 establish the style, components, ports, connectors, roles, and attachments at each level in the architectural hierarchy. For each level below the system level, transform HH_7 establishes the bindings necessary to link the sub-architecture with its parent architecture. Each transform is explained in greater detail in the following paragraphs.

4.4.1 Transform HH_0: The Architectural Model.   The first transformation establishes the root node in the AST, called the `ArchModel`, that encompasses all architectural information pertaining to the system. This transform also creates nodes to represent the basic types (*component, connector, port,* and *role*) as children of `ArchModel`. These classes are abstract classes in the sense that their sole purpose is to *type* the components, connectors, ports, and roles that will be defined by the architecture. Finally, it establishes a `Configuration` node (using the *package* construct) as a child of `ArchModel` to serve as the

root for the system level architecture itself. As such, `Configuration` has as children all the architectural elements of the system (e.g., the specific components, connectors, ports, roles, attachments, etc., that make up the system level architecture) and a *style* node that defines the style implemented by the configuration. The components at the system level configuration can be aggregate or primitive components. Each aggregate component, modeled with the *package* construct, represents the root node of a new level in the architecture hierarchy. In addition to the standard fare of components, connectors, ports, roles, and attachments, all aggregate component nodes have as a child a *style* node and all the *bindings* that map a component's internal representation to its external representation.

The HH_0 transform is defined in the following way:

Transform HH_0:

1. *Instantiate one package node in the model as a child of `SysModel`. Name the new package node `ArchModel`.*

2. *Instantiate one package node in the model as a child of `ArchModel`. Name the new package `Configuration`.*

3. *Instantiate one class node in the model as a child of `ArchModel` for each of the following elements: style, component, connector, port, and role. Name the nodes `Style`, `Component`, `Connector`, `Port`, and `Role`, respectively.*

4. *Return `SysModel`.*

The results of this transform are graphically portrayed by Figure 32.

4.4.2 Transform HH_1: The Style. This transform is responsible for creating the node in the AST that represents the architectural style implemented by the input configuration[3]. The transform accepts the `Configuration` or `Conf_cname` node as input (where Conf_cname is the name of an aggregate component)[4] and is defined in the following way:

---

[3]Note that the configuration can either be the system level `Configuration` or a lower level configuration defined by an aggregate component `Conf_cname`.

[4]To simplify the transform definitions, the input node in each transform will be referred to as `Inode`.
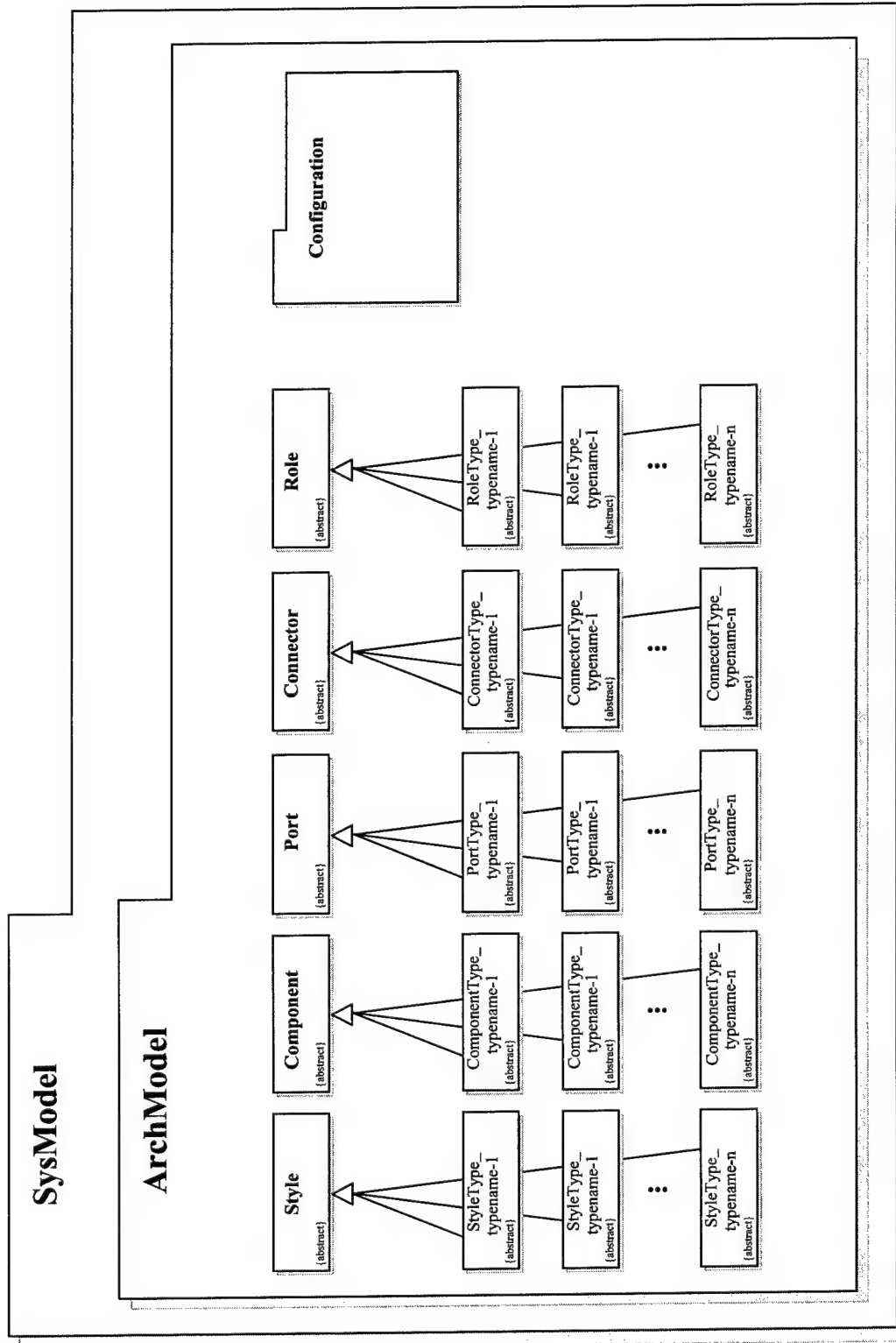
Figure 32.   Architecture model after Transform HH_0.

70
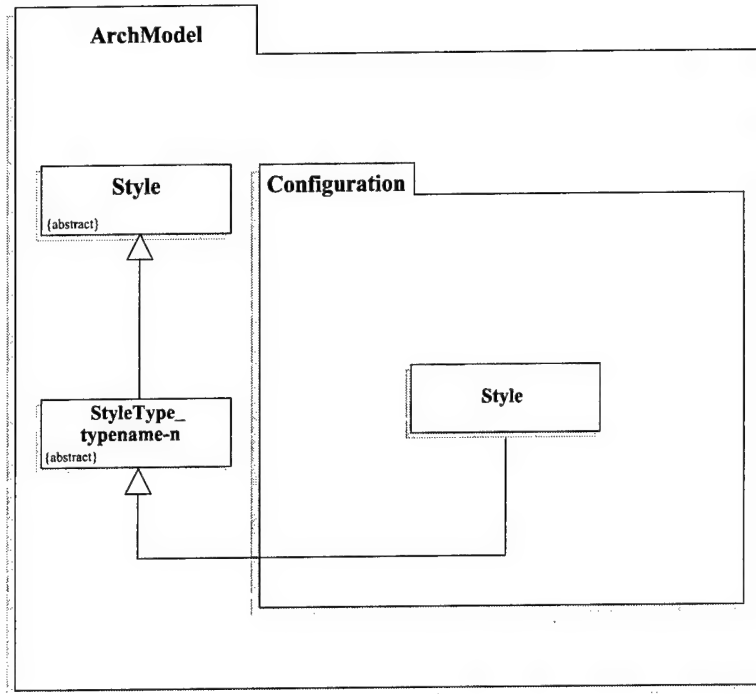
Figure 33.   Architecture model after Transform HH$_1$.

Transform HH$_1$:

1. *Instantiate one class node in the AST as a child of* Inode *to represent the architectural style implemented. Name the node* Style. *Set the appropriate* StyleType_typenamen *class, created as a node under* ArchModel, *as the superclass of* Style.

2. *Return* Inode.

The results of this transform are graphically portrayed by Figure 33.

4.4.3 Transform HH$_2$: The Components.   This transform is responsible for creating the elements in the model that represent components in the input configuration (i.e., Inode). The transform accepts Inode as input and is defined in the following way:

Transform HH$_2$:

1. *For each component defined during architectural design, instantiate one class node as a child of* Inode, *naming each new node* Comp_cname, *where*

71
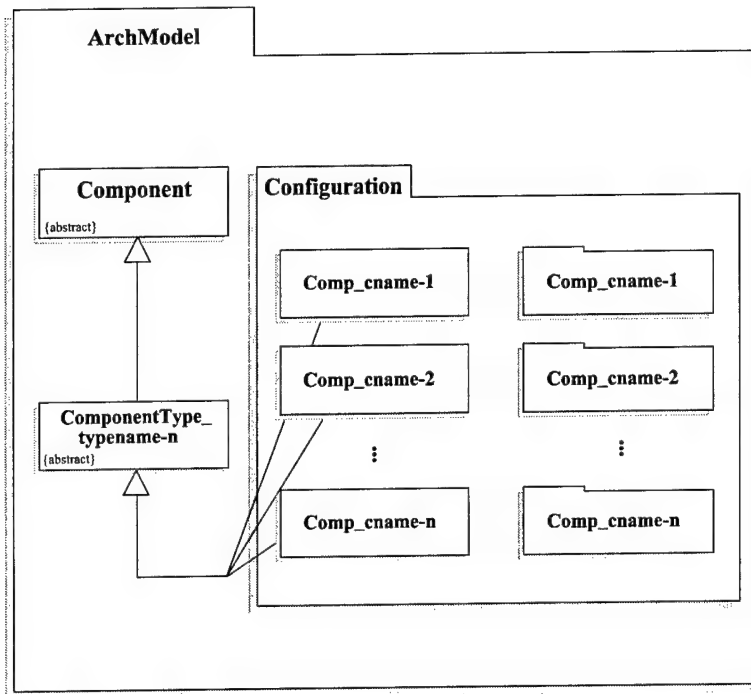
Figure 34.   Architecture model after Transform HH$_2$.

cname is the name given to the component by the architect during archi-
tectural design. Set the Component class as superclass of Comp_cname.

2. In addition, for each aggregate component (i.e., a component consisting
   of lower-level components) defined during architectural design, instantiate
   one package node as a child of Inode, naming each new node Conf_cname,
   where cname is the name given to the component by the architect during
   architectural design[5].

3. Return Inode.

The results of this transform are graphically portrayed by Figure 34.

4.4.4   Transform HH$_3$: The Component Ports.   This transform is responsible for
creating the nodes in the AST that represent component *ports* in the input configuration.
The transform accepts Inode as input and is defined in the following way:

---

[5]Note that for each aggregate component, there exists a package node and a class node representing
that component. This is necessary to allow both typing the component by making it a subclass of Component
and adding children to the component to reflect its container nature.

72

Transform HH₃:

1. *For each port defined during architectural design, instantiate one class node as a child of `Inode` to represent the port. Make the new port class an element of the aggregate class that models the associated architectural component, using whatever mechanism the modeling language provides for aggregation. Name each new port node `Port_pname`, where pname is the name given to the port by the architect during architectural design. Set the `Port` class as superclass of the `Port_pname` class.*

2. *Return `Inode`.*

The results of this transform are graphically portrayed by Figure 35.

4.4.5 Transform HH₄: The Connectors. This transform is responsible for creating the nodes in the AST that represent connectors in the input configuration. The transform accepts `Inode` as input and is defined in the following way:

Transform HH₄:

1. *For each connector defined by the architect during architectural design, instantiate one class node as a child of `Inode`, naming each node `Conn_cname`, where cname is the name given by the architect during architectural design. Set the `Connector` class as superclass of the `Conn_cname` class.*

2. *Return `Inode`.*

The results of this transform are graphically portrayed by Figure 36.

4.4.6 Transform HH₅: The Connector Roles. This transform is responsible for creating the nodes in the AST that represent roles for each connector in the input configuration. The transform accepts `Inode` as input and is defined in the following way:

Transform HH₅:

1. *For each role defined by the architect during architectural design, instantiate one class node as a child of `Inode`, naming the node `Role_rname`,*
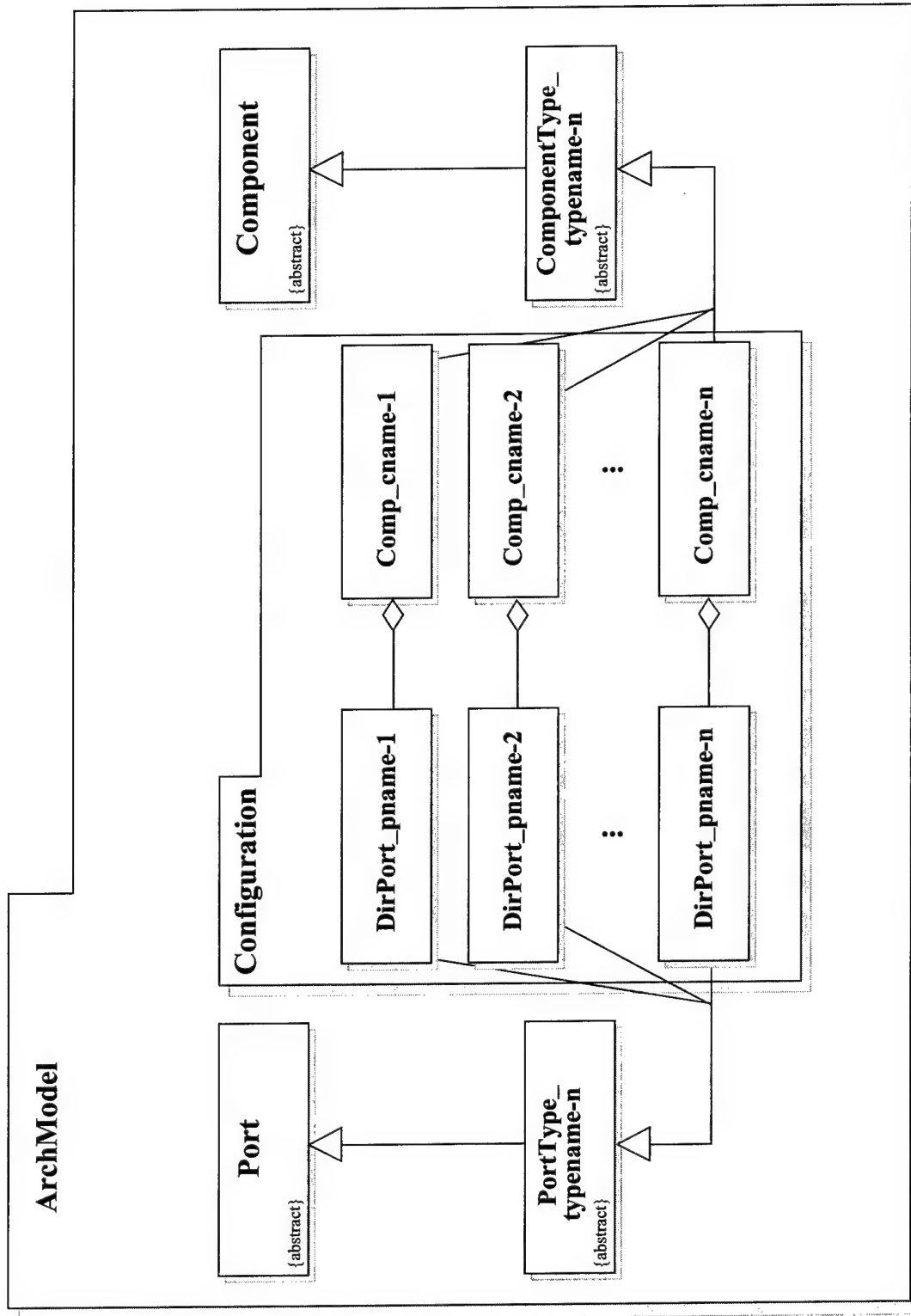
Figure 35. Architecture model after Transform HH$_3$.

74
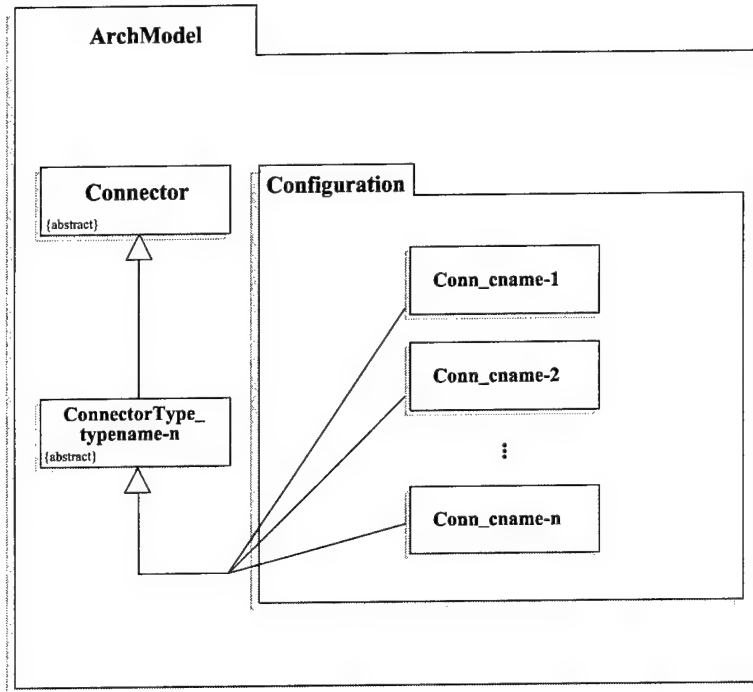
Figure 36.   Architecture model after Transform HH$_4$.

> *where rname is the name given the role by the architect during architec-*
> *tural design. Set class* `Role` *as superclass of the* `Role_rname` *class.*
>
> *2. Return* `Inode`*.*

The results of this transform are graphically portrayed by Figure 37.

4.4.7   Transform HH$_6$: The Port-to-Role Attachments.   This transform is respon-
sible for creating the nodes in the AST that represent attachments in the input configura-
tion. The transform accepts `Inode` as input and is defined in the following way:

Transform HH$_6$:

> *1. For each attachment defined by the architect during architectural design,*
> *instantiate one association node as a child of* `Inode`*, naming the node*
> `Port_pname_attaches_to_Role_rname` *where* `Port_pname` *is the name*
> *of the port involved in the attachment and* `Role_rname` *is the name of the*
> *role involved in the attachment.*

75
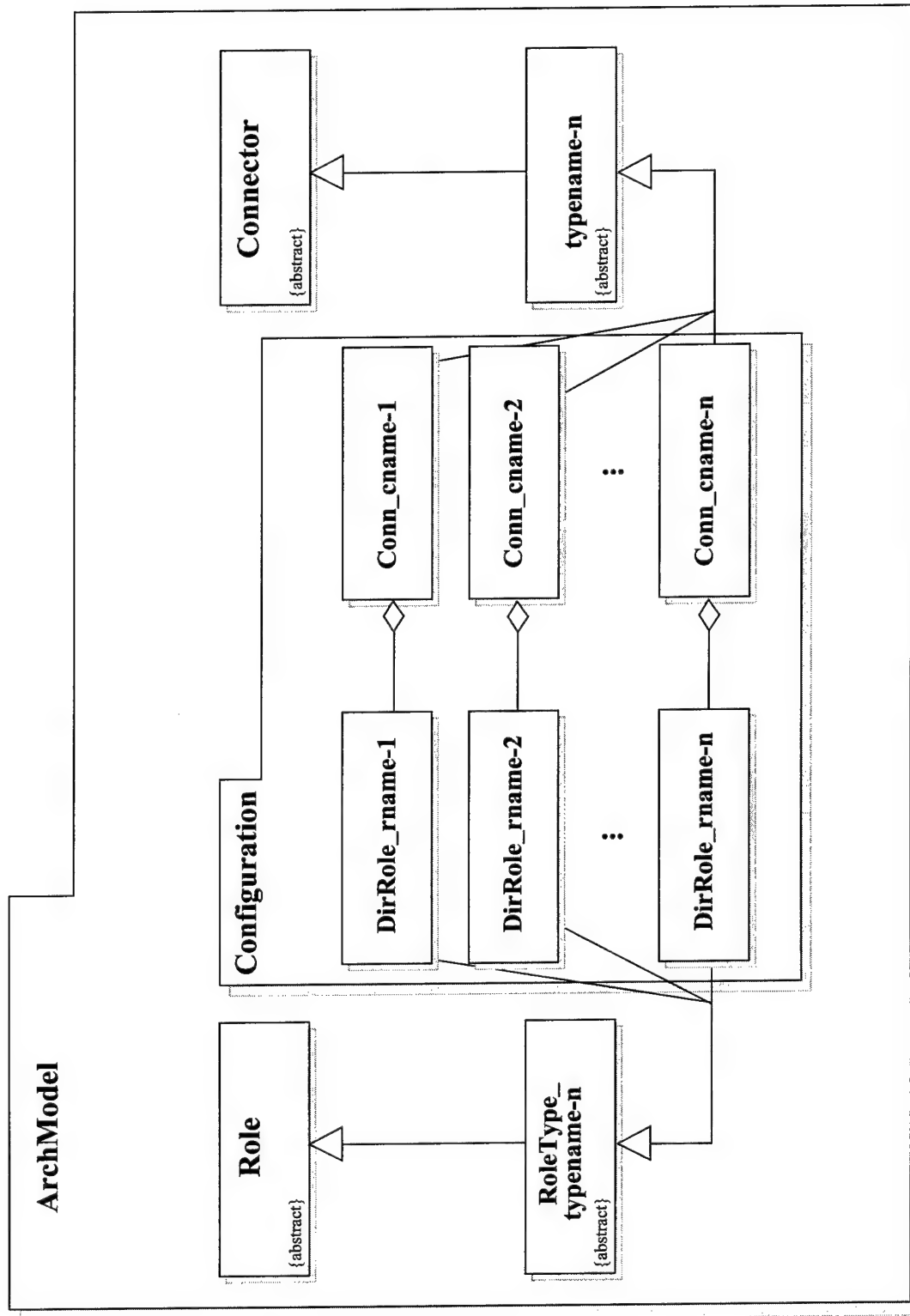
Figure 37.   Architecture model after Transform HH$_5$.

76

2. *For each attachment created in the previous step, link one end of the attachment to the port named* `Port_pname` *and the other end to the role named* `Role_rname` *as specified by the architect during architectural design.*

3. *Return* `Inode`.

The results of this transform are graphically portrayed by Figure 38.

4.4.8   Transform HH₇: The Port-to-Port Bindings.   This transform is *only* run for level-n configurations where $n > 1$. It is responsible for creating the nodes in the AST that represent bindings from the ports in the configuration to the ports of its parent component `Conf_cname` (see paragraph 4.3 and Figure 30 for a review of the role bindings play in the architecture). The transform accepts `Inode` as input and is defined in the following way:

Transform HH₇:

1. *For each binding specified by the architect during architectural design, instantiate one association node as a child of* `Inode` *to represent the attachment, naming the association* `Port_pname₁_binds_Port_pname₂`, *where* `Port_pname₁` *is the name given to a port in the configuration and* `Port_pname₂` *is the name of a port in the parent component to which* `Port_pname₁` *is bound as defined by the architect during architectural design.*

2. *Return* `Inode`.

The results of this transform are graphically portrayed by Figure 39.

4.5   Exporting Architectures to Architectural Interchange Languages

The bulk of this research focused on producing a methodology for broadly classifying software architectures and then formally modeling two of the four architectural classes in a formal, object-oriented transformation system. In this final section, an approach

77
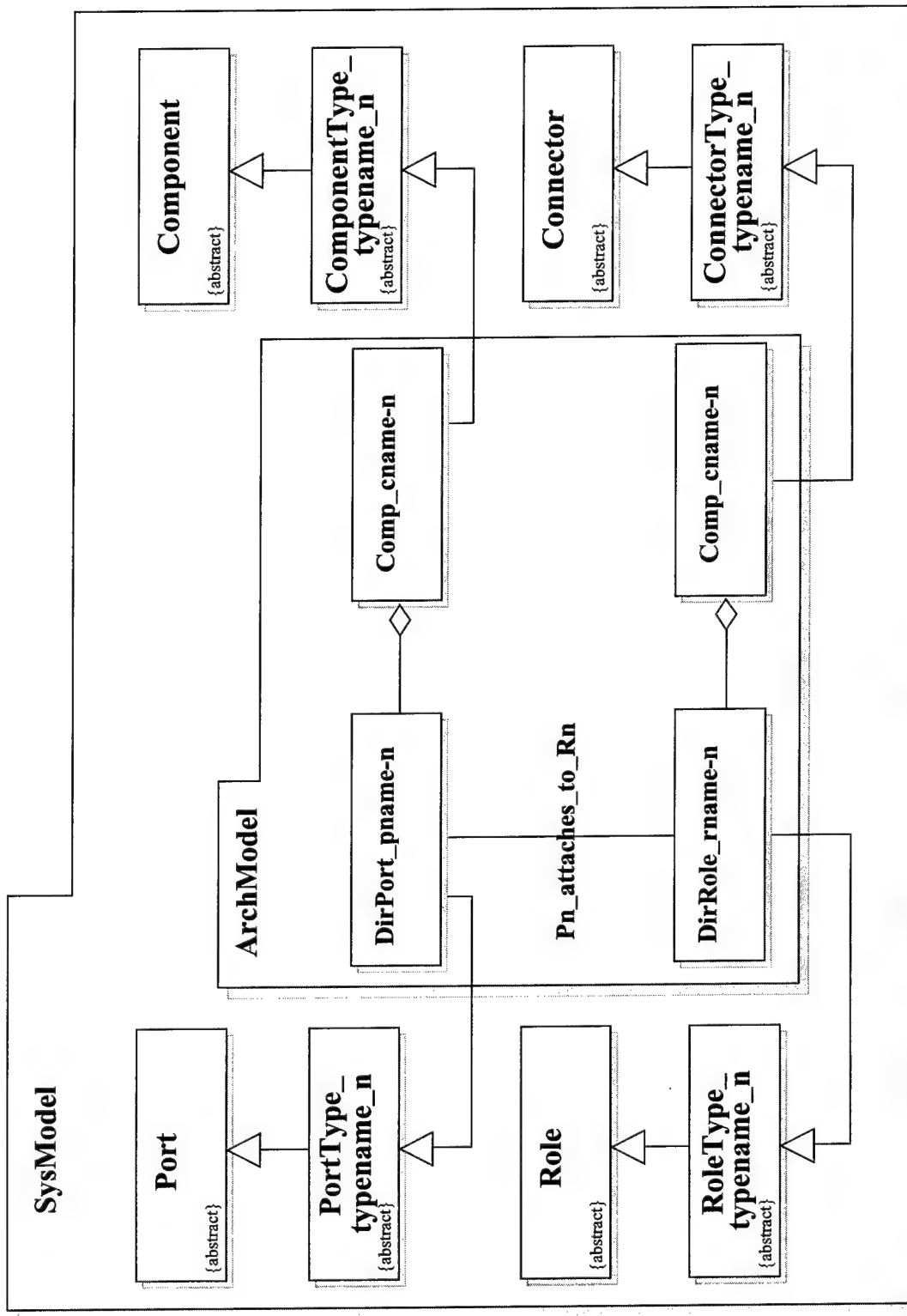
Figure 38.   Architecture model after Transform HH$_6$.

78
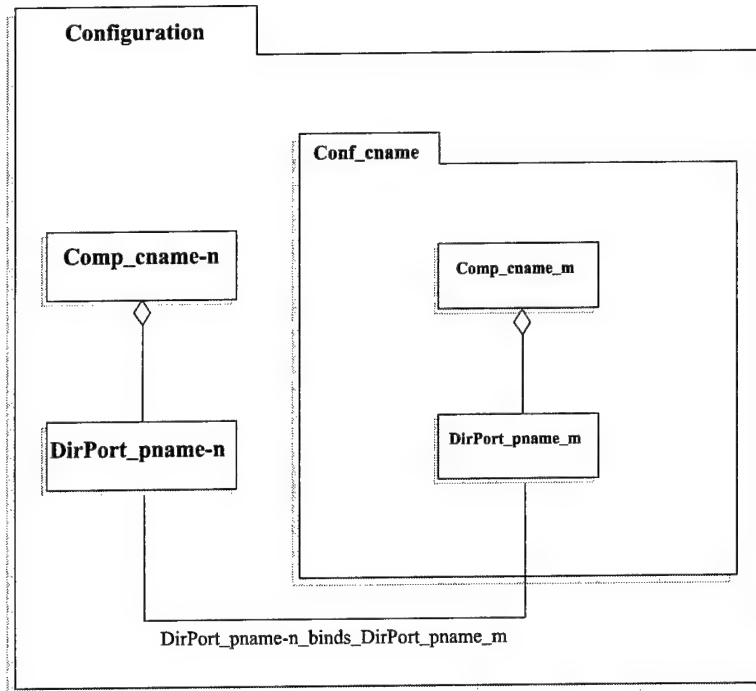
Figure 39.   Architecture model after Transform HHL$_7$.

is presented for extracting the architecture, once in a design AST, to a language fit for architectural interchange between a variety of formal software design environments.

4.5.1 Scope of the Architecture Export Method.   Recalling from the background section, architectural interchange languages attempt to provide a *fixed vocabulary* for expressing software architectures. Such a language defines a canonical syntax sufficient to transfer the essence of an architectural design from one environment to another. Such a language is often called an *intersection* language. The developers of the Acme interchange language went further by incorporating hooks into the language whereby the canonical form representing the essential aspects of the architecture is supplemented (optionally) with *property* blocks containing non-canonical constructs. These properties are then carried along as baggage through the Acme interchange to a foreign environment that may disassemble and make proper use of them. A language that attempts this is termed a *union* language. Acme is both a union and an intersection language [13].

This research limited its scope to the union-oriented aspects of the Acme language. In other words, the foundation it provides is limited to exporting those abstract design elements (e.g., packages, classes, and associations) in a design AST that map directly to the Acme constructs common across ADLs (e.g., systems, components, and bindings). The method for exporting these design elements to Acme surface syntax is described in the following section.

4.5.2 Mapping Architectural Design Elements to Acme Language Constructs. In her master's thesis, Noe demonstrated the ability to infer an architecture from an object-oriented analysis model and produce Acme surface syntax. With the methodology already presented for creating an architecture using an *architecture assistant* and populating a formal, object-oriented design AST, one can now export the architectural design, at any time in the life of the software system, to an Acme surface syntax.

The goal of an algorithm designed to perform such an exportation is to generate correct Acme surface syntax for the architecture. A simple system described in Acme surface syntax is shown in Figure 40.

```
System sample-system = {
   Component A = {
      Port Out;
      Port In;
      Properties { Aesop-style : style-id = pipe-filter;
                   Unicon-style : style-id = pf;
                   source-code : external = ''LIB/a.java''
      };
   };
   Component B = {
      port Send;
      port Receive;
   };
   Connector C = {
      role Acoming;
      role Agoing;
      role Bcoming;
      role Bgoing;
   };
   Attachments {
      A.Out to C.Agoing;
      A.In to C.Acoming;
      B.Out to C.Bgoing;
      B.In to C.Bcoming;
   };
};
```

Figure 40.   A sample system in Acme syntax.

To effect the exportation, the algorithm must adhere to the following mappings between object-oriented design AST elements and Acme program constructs.

Design AST Node⟹Acme Construct:

1. **Configuration(package)⟹System Map:**

   For every **Configuration** package node encountered in the AST, produce an equivalent **System** declaration in Acme.

2. **Component(class)⟹Component:**

   For every **Component** class node encountered in the AST, produce an equivalent **Component** declaration in Acme.

3. **Connector(class)⟹Connector:**

   For every **Connector** class node encountered in the AST, produce an equivalent **Connector** declaration in Acme.

4. **Port(class)⟹Port:**

   For every **Port** class node encountered in the AST, produce an equivalent **Port** declaration in Acme.

5. **Role(class)⟹Role:**

   For every **Role** class node encountered in the AST, produce an equivalent **Role** declaration in Acme.

6. **Attachment(association)⟹Attachment:**

   For every **Attachment** association node encountered in the AST, produce an equivalent **Attachment** instance in an Acme **Attachment** block.

7. **Component(package)⟹Representation:**

   For every **Component** package node encountered in the AST, produce an equivalent **Representation** declaration in Acme.

8. **Binding(association)⟹Binding:**

   For every **Binding** association node encountered in the AST, produce an equivalent **Binding** instance in an Acme **Binding** block.

9. **Style(class)⟹Style:**

   For every **Style** class node encountered in the AST, produce an equivalent **Style** declaration in Acme.

10. **ComponentType(class)⟹Template:**

    For every **ComponentType** class node encountered in the AST, produce an equivalent **Template** declaration in Acme.

11. **ConnectorType(class)⟹Template:**

    For every **ConnectorType** class node encountered in the AST, produce an equivalent **Template** declaration in Acme.

4.6 Summary

Creating hierarchically heterogeneous architectures and inserting them into an AST requires more than simply inferring the architecture from an analysis model. An architectural modeling tool that facilitates the application of one or more architectural styles to an analysis model is required. Once an architecture is composed, the results can be transformed to an object-oriented design AST by applying a set of style-specific transforms. With architectural models explicitly reflected in a design AST, systems engineers are able to extract and reason over the architecture by using analysis tools native to the environment or by exporting the architecture to other analysis tools via architectural interchange languages.

83

V. Demonstration

5.1 Overview

This chapter summarizes the results of a trial run of the methodology. The trial run, or demonstration, was conducted in the AWSOME transformational environment. AWSOME is a representative example of the target environment and provided fertile ground for this demonstration.

For the purposes of demonstrating the methodology, a sample problem was needed. The problem had to be simple enough to be solved in a limited amount of time, but flexible enough to be able to apply both an NM and HH architecture during the demonstration. The problem profile became a simple, object-oriented, client-server application.

5.2 An Architecture Waiting to Happen

Bā′ bəl is an online multilingual dictionary developed by the author and an associate to satisfy a distributed operating systems course requirement at the Air Force Institute of Technology. The intent of the development project was to expose students to distributed client-server application development techniques. The purpose of the *Bā′ bəl* system itself is to provide a way to quickly get a foreign language translation of an English word or phrase using distributed, and perhaps web-based, technology. The system requires the facilities necessary for a user to enter an English word or phrase, the name of a target language, and a command to proceed with translation (see Figure 41). Upon receiving valid input, the system returns the translation of the word or phrase written in the Western alphabet, the translation as it is represented in the target language (i.e., the native alphabet), and a sound file that demonstrates the correct pronunciation of translation in the native dialect.

Bā′ bəl was eventually implemented as a distributed client-server system written in Java™ using Java™ remote message invocation (RMI). The server component was implemented using Oracle®. Because the *Bā′ bəl* project presented a simple distributed client-server architecture and was implemented using object-oriented development tech-

84
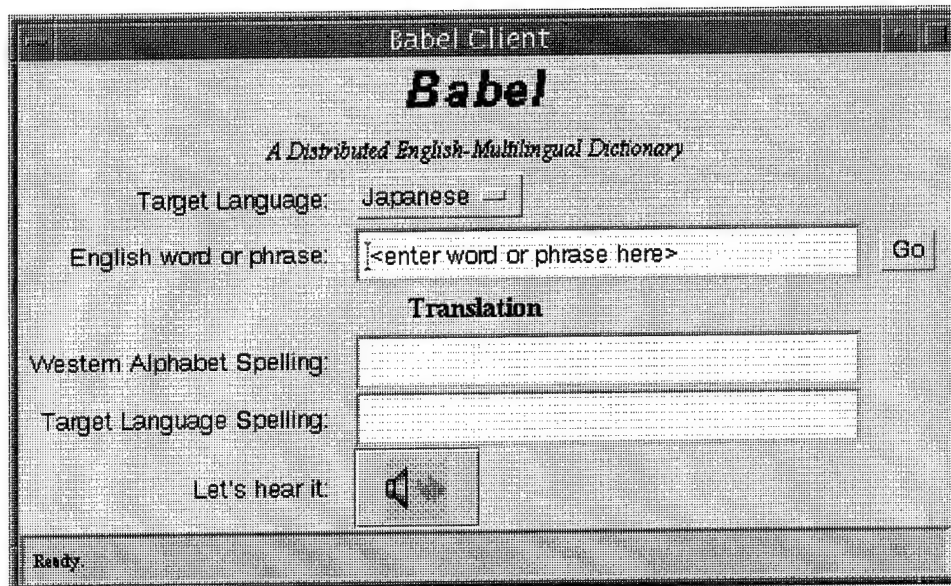
Figure 41. **Bā′ bəl** graphical user interface.

niques, it was particularly well-suited to demonstrating the methodology presented in this research.

This thesis includes as Appendix A the formal **Bā′ bəl** specification expressed in Z syntax. In a completely operational transformation system, this formal Z specification would have been parsed and loaded into an AWSOME AST as an analysis model. However, at the time of this writing there was no such Z-to-AWSOME parser[1]. Therefore, as a preliminary step in the demonstration, a program was developed in Java to directly create the **Bā′ bəl** analysis model as an AWSOME AST in a manner that is consistent with the Z specification. In addition, a simple Java applet was created to facilitate the demonstration itself. The applet provides a menu of choices, as shown in Figure 42, for building and manipulating an AWSOME AST. Menu items 1 and 2 instantiate the AST and populate the analysis model.

With an AWSOME AST—including a `SysModel` and an `AnalysisModel`—as a starting point, the two architectural variations were produced.

---

[1] In an earlier version of AWSOME, there was a Z-to-DOM (domain object model) parser

```
Applet Viewer: BabelSystem.ArchInterface        _ □ ☒
Applet

    Menu

    1 - Create new Awsome model

    2 - Generate analysis model

    3 - Generate Object-Oriented architectural model

    4 - Generate OO Client-Server architectural model

    5 - Print the AST (stdout)

    6 - Export the architectural model to Acme

    7 - Initialize the AST (i.e., empty it)


            Enter choice:  [              ]

Applet started.
```
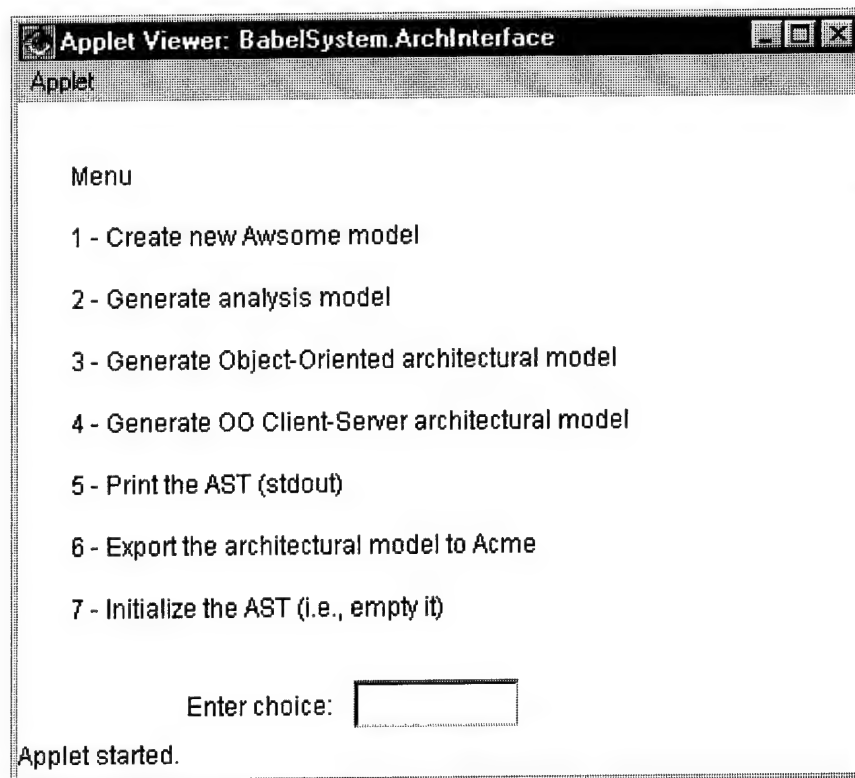
Figure 42.   Architecture demonstration applet interface.

5.3 Modeling an Object-Oriented **Bā′ bəl** Architecture

For the first part of the demonstration, the target architecture for **Bā′ bəl** was the non-hierarchical object-oriented style. This is the class and style of architecture that AWSOME implicitly builds by default. The purpose of the demonstration was to show the feasibility of explicit modeling via automatic transformation.

A program was written to walk the analysis model and automatically apply the transforms discussed in 3.6. The tree-walking exercise itself offered no new or surprising results—it had, in fact, been performed before in a similar environment by Noe. The transformation of the analysis model to an explicit *architectural* model, however, is new and was accomplished successfully using transforms NM_0 through NM_5, also with unsurprising results. Appendix C shows the AWSOME surface syntax for the object-oriented model created by the transformation process. Appendix D shows the Acme surface syntax generated by the Acme export process.

5.4 Modeling a Client-Server **Bā′ bəl** Architecture

The second target architecture for **Bā′ bəl** was a client-server style with object-oriented clients and servers. As previously described, the production of such an architecture from an object-oriented analysis model requires additional information, perhaps best provided through an architectural modeling utility. In the absence of such a utility, a program was written in Java that simulated the design decisions imposed by an architect and built the target architecture as an AWSOME AST. For example, for the **Bā′ bəl** system, the program assumed that the architect decided to group *Comp_LookupPButton*, *Comp_AddEntryPButton*, and *Comp_BabelInterface* together to form the client component *Comp_BabelClient*. Knowing this, it creates a *Comp_BabelClient* aggregate class with three subcomponents (i.e., *Comp_LookupPButton*, *Comp_AddEntryPButton*, and *Comp_BabelInterface*).

This transformation required the introduction of two new package nodes in the AWSOME AST as children of `ArchModel`. The first of the two nodes, called `Style` was reserved for the inclusion of architectural style information pertinent to the architecture

87

being modeled—it was unused during this implementation. The second of the two nodes was the `Configuration` node. The configuration node provided a root for the architectural configuration of the system apart from the nodes used to represent the architectural styles and element types used in the configuration.

Transformation HH_0 initiated the transformation process by correctly appending to the AST root `SysModel`, a node called *ArchModel_Babel_CS* that would serve as the root of the new architectural AST. In addition, it created all architectural style and port/role type nodes that would be used to *type* the elements of the architecture. For example, not only did the transformation create a `component` node, it also created three subclasses of component called `ComponentType_Object`, `ComponentType_Client`, and `ComponentType_Server`. As subclasses of the `Connector` node, the transformation created `ConnectorType_MethodCall` and `ConnectorType_RPC`. These nodes reflect the heterogeneous (i.e., object-oriented client-server) nature of the architecture being modeled for *Bā' bəl* .

Transforms HH_1 through HH_6 created the top level in the system architecture. Based on the decisions of the architect, the transformations created a `Style` node as a subclass of `StyleType_ClientServer` (HH_1), component nodes for each client and server identified at the system level and their respective ports (HH_2 and HH_3), connector nodes to represent directional RPC between the clients and servers and their respective roles (HH_4 and HH_5), and attachments linking the client and server component ports to the RPC connector roles (HH_6). The rules for creating these architectural entities for a client-server configuration were quite different, so the transformations for this exercise borrowed little from the first transformation of an object-oriented architecture. For instance, in the object-oriented transformation, a port is created for each and every event received or sent by each and every object component. In the client-server transformation, the only ports required were `Port_ClientOut` and `Port_ClientIn` (and a similar pair for the server component) to represent the bidirectional nature of the RPC connection. The single `Conn_RPC` connector had four roles with which to attach itself to incoming and outgoing RPC communication.

The second iteration of transforms $\mathbf{HH}_1$ through $\mathbf{HH}_6$ created the second level in the configuration for the `Comp_BabelClient` aggregate component. This sub-architecture instantiated the object-oriented architectural style, as opposed to the system level client-server style. The most noteworthy difference at this level was the inclusion of *bindings*, created by transform $\mathbf{HH}_7$, that tied the dynamics of the sub-architecture with that of the system level architecture.

Like the transformations for the NM-class $\mathbf{B\bar{a}' b\partial l}$ architecture, these transformations executed as expected and produced explicit software architecture models as AWSOME ASTs. The results of the transformation process are included as Appendix E. The final objective in the demonstration was to show how the architecture can be extracted from the design AST into a language that allows interchanging the architecture to and from dissimilar modeling environments.

5.5 Exporting the $\mathbf{B\bar{a}' b\partial l}$ Architecture to Acme

Explicitly modeling a software architecture in a transformational environment provides a way to constrain the lower-level design transforms and opens the door to more complex system architectures. An alternative benefit of explicit architectural modeling is the ability to extract the architecture from the system and analyze it using tools and techniques that do not readily support the local modeling language.

In this part of the implementation, the hierarchically heterogeneous architecture was exported to the Acme interchange language. The demonstration involved developing a small Java applet to support the operations. As previously noted, the applet supported the creation of object-oriented as well as object-oriented client-server architecture models as AWSOME ASTs. The applet also provided a mechanism for generating Acme code from the AST. The transforms themselves were written as the `WsAcmeHHVisitor` class in the `WsVisitors` package. The visitor class extends the `WsCodeVisitor` class and implements `WsVisitor`. The visitor class includes a visit method for each of the three AWSOME constructs used by the architectural model—`WsPackage`, `WsClass`, and `WsAssociation`.

89

The visit method for `WsPackage` had to account the four variations of a package that can exist in an architecture model—the `SysModel` package, the `ArchModel` package, the `Configuration` package, and all aggregate component (i.e., `Comp_BabelClient`). When the method is called, it is passed a node that is one of these four types. The method determines the type based on the value of the `WsIdentifier` of the package and generates the appropriate Acme code.

Similarly, the visit method for `WsClass` had to account for the four kinds of classes that can occur in the architecture—components, ports, connectors, and roles. To determine the appropriate Acme code generation scheme, the visit method keys on the value of the `WsIdentifier` of the node passed in as a parameter.

The visit method for `WsAssociation` was concerned with two types of associations— architectural *attachments* and *bindings*. As before, the method looks at the value of the `WsIdentifier` and, based on that value, produces appropriate Acme surface syntax.

The Acme code generator, once developed, achieved the desired results—correct Acme surface syntax sufficient for interchange with other modeling environments. The output from the code generator for the HH-class (client-server) architecture is included as Appendix F.

5.6 Summary

The architectural modeling methodology presented was demonstrated using a simple object-oriented client-server application called **Bā′ bəl** . The demonstration involved three key objectives. The first objective was to successfully parse the analysis AST and generate the appropriate abstract syntax for an NM-class object-oriented architecture for the **Bā′ bəl** system. This objective was fully met. The second objective was to successfully represent an object-oriented client-server (i.e., HH-class) architecture for **Bā′ bəl** based on the same analysis model. An AST-generation program was written in lieu of an architectural modeling tool. This objective was fully met. The final objective was to successfully export the HH-class architecture from the design AST to an Acme surface syntax. This objective was fully met.

VI. Conclusions and Recommendations

6.1 Conclusions

Several conclusions were made following this research. First, object-oriented analysis models are not a source of high-level design information. In fact, the objective is to limit, as much as possible, the amount of design that goes into the requirements specification process. Although total elimination of design information may not be practical, or desired, in a transformational environment, keeping the specification relatively free of lower-level design and implementation constraints helps prevent the unwitting elimination of viable alternative solutions. The appropriate place to assert and examine design information, including high-level system architectures, is in post-specification formal models (e.g., designs).

A second observation is the sheer breadth of architectural design alternatives in the field. There is a community focused on architecture and design patterns and pattern languages, a group dedicated to the use of architectural frameworks in software development, and those like the Software Engineering Institute who key on architectural styles, software qualities, and process models. It is not altogether clear that a single approach to software architectures is sufficient for all development paradigms. It does seem important, however, that in a given environment there should be a way to assert high-level design decisions, produce meaningful software system architectures, and analyze these abstract solutions for their efficacy prior to continuing the development process.

Finally, semi-automated software transformation systems cannot produce software solutions without the involvement of highly trained and educated software professionals. There must be a mechanism for introducing the decisions made by these facilitators into the transformational process. Software architectural modeling is an appropriate mechanism whereby design constraints can be placed on a software solution.

6.2 Contributions

The significant contributions of this research to the field of software engineering and the area of formal transformation systems are:

91

- it presents a modified version of the transformational process model with an increased emphasis on explicit software architecture modeling and representation,

- it defines a taxonomy of architectural *classes* for use in those environments capable of producing hierarchically heterogeneous architectures,

- it demonstrates the feasibility of explicitly modeling the architectures of software systems in object-oriented formal transformation systems,

- it expands the notion of architectures in object-oriented environments to include style-based hierarchically heterogeneous configurations,

- it redirects the focus of architecture extraction methods from analysis models to design models, and

- it discusses the utility, feasibility, and desirability of verifying the conformance of software architectures to well-delineated classes and styles of architecture.

6.3 Recommendations for Future Work

The cost of explicitly composing and formally modeling software architectures is only justified if the architectural models can then be used to improve the designs of the systems for which they were created. Therefore, there must be a way to extract the architectures and analyze them to determine their ability to effectively solve formally specified problems. Examples of such analyses are port-component consistency, port-role compatibility, connector deadlock freedom (Wright), and satisfaction of ordering relations by run-time trace sets (Rapide). There is much talk in the research community about third-party analysis tools and interchange languages available for this purpose, but very little has been done to show, convincingly, that this is feasible. One future research possibility is the exploration of such analyses for the purpose of defining a base set, along with the tools that support them, that can be easily applied to an architectural model.

Second, the modified process model suggested in this research depends on the ability to appropriately *profile* a system based on its quality attributes and to select an appropriate combination of architectural styles based on this profile. However, capturing, quantifying, and formalizing the quality attributes is an area that has not been adequately explored.

It has always been difficult to quantify the "-ilities" required by a system—now we are recognizing the dependence of good architectural modeling on this quantification. Therefore, quantifying and formally modeling the non-functional quality attributes introduced by stakeholders in the development of software systems and leveraging their availability to produce software systems that meet the often-overlooked demands place on them is an area ripe for future research.

A third area of possible future research is the notion of a graphical architecture composition utility. Such an application would provide a way to graphically compose complex architectures over previously specified analysis models, thereby hiding the formalisms that make such modeling difficult. The tool would be supported by an architectural domain model that would provide the knowledge necessary to correctly apply architectural styles to the problem. Also, the tool would provide a way to export a formal representation of the architecture once the architect is content with the design.

Finally, the architecture of a software system earns its place in the process model by providing a way to constrain the down-stream transformation to code. Therefore, an area that requires significantly more research is that of composing the newly modeled architecture with the analysis model to produce a robust low-level design. The analysis model by itself carries very little, if any, design information, but provides essential information for the creation of low-level design entities; the architecture provides the design inputs required to produce an appropriate low-level design for the target system. This process is alluded to, and vaguely described, in this research, but is essential to the complete inclusion of software architecture in the formal transformation process.

6.4 Summary

The primary contribution of this research is that it provides a way to explicitly, and formally, model software architecures using an object-oriented modeling language native to a transformation system. In addition, it proposes a modified process model, based on explicit architectural modeling, to generate a large class of hierarchically heterogeneous software architectures in a formal, object-oriented, transformational development environment. These high-level designs can be analyzed—using analysis tools from a variety of de-

velopment environments—for their ability to satisfy the critical software quality attributes espoused by key stakeholders. Additionally, the process model provides a convenient way to introduce design decisions into the transformation process, thereby constraining lower-level transforms and facilitating the production of more complex object-oriented software systems. The research targeted object-oriented environments that typically lack an architectural emphasis and provided a new system-level perspective on the transformational development process model. It laid a solid foundation for developing and analyzing hierarchically heterogeneous architectures in transformational environments and established a launch pad for future work on the topic.

Appendix A. *Z Analysis Model for* **Bā′ bəl**

Bā′ bəl is an interactive multilingual translation dictionary. It accepts a target language and an English word or phrase as input (e.g., < "Japanese", "dog" >)and returns the foreign language translation of the English word or phrase (e.g., "inu").

[Note: In the original system, the translation is returned in three different forms: a US-English textual representation of the translation, a foreign textual representation of the translation, and an audio clip that demonstrates the proper pronunciation of the translation. For example, on receiving a request to provide a Japanese translation for the English phrase "good morning," the system would respond with "ohayou gozaimasu" (the US-English textual representation), the Japanese textual representation for "ohayou gozaimasu" (using a unicode font), and a sound clip of the proper pronunciation of "ohayou gozaimasu" in the modern Japanese vernacular. However, for the purposes of this thesis, the system will be simplified to simply return only the US-English textual representation of the translation.]

Object Name: Word

Object Number: 1

Object Description: This object models the notion of a dictionary word.

Date: 2/5/00
History: 2/5/00: Original

Author: Williams

Superclass: None

Components: None

Context: None

Attributes:

spelling	seq CHAR	English spelling of Word
origin	LANGUAGE	Origin of Word

Constraints:

None

Z Static Schema:

$LANGUAGE ::= English \mid Japanese \mid German$

$$
\begin{array}{|l}
\hline
\textit{Word} \\
\hline
spelling : seq\ CHAR \\
origin : LANGUAGE \\
\hline
\end{array}
$$

Object Name: EngWord

Object Number: 2

Object Description: This object models an English word in the dictionary.

Date: 2/5/00

History: 2/5/00: Original

Author: Williams

Superclass: Word

Components: None

Context: None

Attributes:

spelling	seq CHAR	inherited from Word
origin	LANGUAGE	inherited from Word

Constraints:

origin = English

Z Static Schema:

$$
\begin{array}{|l}
\hline
\_EngWord_____ \\
\quad Word \\
\hline
\quad origin = English \\
\hline
\end{array}
$$

Object Name: NihWord

Object Number: 3

Object Description: This object models a Japanese word in the dictionary.

Date: 2/5/00

History: 2/5/00: Original

Author: Williams

Superclass: Word

Components: None

Context: None

Attributes:

spelling	seq CHAR	inherited from Word
origin	LANGUAGE	inherited from Word

Constraints:

origin = Japanese

Z Static Schema:

```
__NihWord_____
  Word
 _____
  origin = Japanese
```

Object Name: GerWord

Object Number: 4

Object Description: This object models a German word in the dictionary.

Date: 2/5/00
History: 2/5/00: Original

Author: Williams

Superclass: Word

Components: None

Context: None

Attributes:

spelling	seq CHAR	inherited from Word
origin	LANGUAGE	inherited from Word

Constraints:

origin = German

Z Static Schema:

$$
\begin{array}{|l}
\hline
\_GerWord\underline{\hspace{8cm}} \\
\quad Word \\
\quad \underline{\hspace{2cm}} \\
\quad origin = German \\
\hline
\end{array}
$$

Object Name: Entry

Object Number: 5

Object Description: This object models an entry in the multilingual dictionary.

Date: 2/5/00

History: 2/5/00: Original

Author: Williams

Superclass: None

Components:

EngWord
NihWord
GerWord

Context: None

Attributes:

eng_word	EngWord	an English word
nih_word	NihWord	the Japanese translation of eng_word
ger_word	GerWord	the German translation of eng_word

Constraints:

eng_word cannot be null

Z Static Schema:

$$
\begin{array}{l}
\underline{\quad Entry\quad}\\
\quad eng\_word : EngWord\\
\quad nih\_word : NihWord\\
\quad ger\_word : GerWord\\
\hline
\quad \#eng\_word.spelling > 0\\
\end{array}
$$

BabelDictionary Structure Definition

Object Name: BabelDictionary

Object Number: 6

Object Description: This object encapsulates the multilingual translation data.

Date: 2/5/00
History: 2/5/00: Original

Author: Williams

Superclass: None

Components: None

Context: None

Attributes:

table   seq Entry   sequence of Entry objects

Constraints:

Entries in table are alphabetized on eng_word

Z Static Schema:

$$\begin{array}{|l}
\hline
\_BabelDictionary _____ \\
table : seqEntry \\
\hline
\forall\ e1, e2 : Entry;\ i, j : Nat \bullet i \leq \#table \wedge j \leq \#table \wedge i < j \wedge \\
e1 = table'(i) \wedge e2 = table'(j) \Rightarrow reduce(concat, e1) < reduce(concat, e2) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\_InitDictionary _____ \\
\Delta BabelDictionary \\
\hline
\#table = 0 \\
\hline
\end{array}$$

BabelDictionary Functional Model

Object: *BabelDictionary*

Process Name: AddEntry

Process Description: Adds the specified entry to the table of entries.

Z Dynamic Schema:

$$
\begin{array}{l}
\underline{\quad AddEntry \quad\rule{7cm}{0pt}} \\
\Delta BabelDictionary \\
entry? : Entry \\
\hline
entry? \in table'
\end{array}
$$

Process Name: FindTranslation

Process Description: Returns a foreign translation of the input English word in the input target language.

Z Dynamic Schema:

$$
\begin{array}{l}
\underline{\quad FindTranslation \quad\rule{5cm}{0pt}} \\
\Xi BabelDictionary \\
translation! : seqCHAR \\
word? : seqCHAR \\
tgtLang? : LANGUAGE \\
\hline
\forall e : Entry \bullet e \in table \wedge e.eng\_word.spelling = word? \Rightarrow \\
\quad (tgtLang? = Japanese \wedge translation! = e.nih\_word.spelling) \vee \\
\quad (tgtLang? = German \wedge translation! = e.ger\_word.spelling)
\end{array}
$$

State Name: START

State Description: Initial startup state.

Z Static Schema:

```
┌─ START ────────────────────────────────────────────────
│ BabelDictionary
├────────────────
│ True
└─────────────────────────────────────────────────────────
```

State Name: Ready

State Description: Ready and waiting for a lookup request.

Z Static Schema:

```
┌─ Ready ────────────────────────────────────────────────
│ BabelDictionary
├────────────────
│ True
└─────────────────────────────────────────────────────────
```

State Name: Busy

State Description: Looking up a request.

Z Static Schema:

```
┌─ Busy ─────────────────────────────────────────────────
│ BabelDictionary
├────────────────
│ True
└─────────────────────────────────────────────────────────
```

Event Name: DoLookup

Event Description: DoLookup received from user.

Z Static Schema:

```
 ┌─ DoLookup ──────────────────────────────────
 │ aWord : seqCHAR
 ├──────────
 │ True
 └─────────────────────────────────────────────
```

Event Name: DoAddEntry

Event Description: DoAddEntry received from user.

Z Static Schema:

```
 ┌─ DoAddEntry ────────────────────────────────
 │ anEntry : Entry
 ├──────────
 │ True
 └─────────────────────────────────────────────
```

Event Name: NotFound

Event Description: Word not found in table.

Z Static Schema:

```
 ┌─ NotFound ──────────────────────────────────
 │ True
 └─────────────────────────────────────────────
```

Event Name: Found

Event Description: Word found in table.

Z Static Schema:

```
 ┌─ Found ─────────────────────────────────────
 │ True
 └─────────────────────────────────────────────
```

Event Name: TransFound

Event Description: Successful find message sent to user.

Z Static Schema:

```
┌─ TransFound ──────────────────────────────────────────
│ result : seqCHAR
├────────────────────────────────────────────────────────
│ True
└────────────────────────────────────────────────────────
```

Event Name: TransNotFound

Event Description: Unsuccessful find message sent to user.

Z Static Schema:

```
┌─ TransNotFound ───────────────────────────────────────
├────────────────────────────────────────────────────────
│ True
└────────────────────────────────────────────────────────
```

State Transition Table:

Current	Event	Guard	Next	Action	Send
START			Ready	InitDictionary	
Ready	DoLookup		Busy	FindTranslation	
Ready	DoAddEntry		Ready	AddEntry	
Busy	NotFound		Ready		TransNotFound
Busy	Found		Ready		TransFound

Object Name: PushButton

Object Number: 7

Object Description: This object models a gui push button.

Date: 2/5/00

History: 2/5/00: Original

Author: Williams

Superclass: None

Components: None

Context: None

Attributes:

name	seq CHAR	names PushButton
status	PBSTATE	models status of push button

Constraints:

None

Z Static Schema:

$PBSTATE ::= Activated \mid Deactivated$

PushButton
$name : seqCHAR$
$status : PBSTATE$

Object Name: TextBox

Object Number: 10

Object Description: This object models a gui text box.

Date: 2/5/00

History: 2/5/00: Original

Author: Williams

Superclass: None

Components: None

Context: None

Attributes:

name	seq CHAR	names TextBox
value	seq CHAR	stores value of TextBox
editable	BOOLEAN	differentiates between editable and non-editable text boxes

Constraints:

None

Z Static Schema:

$$
\begin{array}{|l}
\hline
\_TextBox _____ \\
\quad name : seqCHAR \\
\quad value : seqCHAR \\
\quad editable : BOOLEAN \\
\hline
\end{array}
$$

Object Name: StatusBar

Object Number: 11

Object Description: This object models a gui status bar used to display messages to the user.

Date: 2/5/00

History: 2/5/00: Original

Author: Williams

Superclass: None

Components: None

Context: None

Attributes:

name	seq CHAR	names StatusBar
value	seq CHAR	stores value of StatusBar

Constraints:

None

Z Static Schema:

$$\begin{array}{l} \underline{StatusBar} \\ \quad name : seqCHAR \\ \quad value : seqCHAR \\ \end{array}$$

Object Name: BabelInterface

Object Number: 12

Object Description: This object models the Babel system gui.

Date: 2/5/00

History: 2/5/00: Original

Author: Williams

Superclass: None

Components: None

Context: None

Attributes:

tgtLang	LANGTYPE	determines target language for translation
e_word	TextBox	captures word to be translated
f_word	TextBox	displays translation of e_word
translate	LookupPButton	generates lookup event when pushed
addEntry	AddEntryPButton	generates AddEntry event when pushed
sBar	StatusBar	displays messages to user during session

Constraints:

e_word.editable must be True
f_word.editable must be False

Z Static Schema:

$$
\begin{array}{l}
\underline{BabelInterface} \\
\hline
tgtLang : LANGTYPE \\
e\_word : TextBox \\
f\_word : TextBox \\
translate : LookupPButton \\
addword : AddEntryPButton \\
sBar : StatusBar \\
\hline
e\_word.editable = True \\
f\_word.editable = False \\
\hline
\end{array}
$$

InitInterface _____
$\Delta BabelInterface$

$\#e\_word.value = 0$
$\#f\_word.value = 0$
$sBar.value = \text{``}Ready.\text{''}$

Object: *BabelInterface*

Process Name: DisplayResult

Process Description: Displays the translation.

Z Dynamic Schema:

$$
\begin{array}{l}
\_DisplayResult _____ \\
\Delta BabelInterface \\
result? : seqCHAR \\
\hline
f\_word.value' = result? \\
_____
\end{array}
$$

Process Name: DisplayError

Process Description: Displays an error message.

Z Dynamic Schema:

$$
\begin{array}{l}
\_DisplayError _____ \\
\Delta BabelInterface \\
\hline
sBar.value = e\_word.value + \text{``} not found in dictionary.\text{''} \\
_____
\end{array}
$$

State Name: START

State Description: Initial startup state.

Z Static Schema:

```
┌─ START ────────────────────────────────────────────────────
│ BabelInterface
├──────────────
│ True
└────────────────────────────────────────────────────────────
```

State Name: Ready

State Description: Ready for an input from an external source.

Z Static Schema:

```
┌─ Ready ────────────────────────────────────────────────────
│ BabelInterface
├──────────────
│ True
└────────────────────────────────────────────────────────────
```

State Name: Waiting

State Description: Waiting for a response from dictionary.

Z Static Schema:

```
┌─ Waiting ──────────────────────────────────────────────────
│ BabelInterface
├──────────────
│ True
└────────────────────────────────────────────────────────────
```

Event Name: Lookup

Event Description: Lookup request received from user.

Z Static Schema:

```
┌─ Lookup ─────────────────────────────────────────────────
│ True
└──────────────────────────────────────────────────────────
```

Event Name: AddEntry

Event Description: AddEntry request received from user.

Z Static Schema:

```
┌─ AddEntry ───────────────────────────────────────────────
│ True
└──────────────────────────────────────────────────────────
```

Event Name: DoLookup

Event Description: DoLookup sent to dictionary.

Z Static Schema:

```
┌─ DoLookup ───────────────────────────────────────────────
│ aWord : seqCHAR
├──────────────────────────────────────────────────────────
│ True
└──────────────────────────────────────────────────────────
```

Event Name: DoAddEntry

Event Description: DoAddEntry sent to dictionary.

Z Static Schema:

```
┌─ DoAddEntry ─────────────────────────────────────────────
│ anEntry : Entry
├──────────────────────────────────────────────────────────
│ True
└──────────────────────────────────────────────────────────
```

Event Name: TransFound

Event Description: Successful find message received from dictionary.

Z Static Schema:

```
┌─ TransFound ────────────────────────────────────────────
│ result : seqCHAR
├──────────────────────────────────────────────────────────
│ True
└──────────────────────────────────────────────────────────
```

Event Name: TransNotFound

Event Description: Unsuccessful find message received from dictionary.

Z Static Schema:

```
┌─ TransNotFound ─────────────────────────────────────────
├──────────────────────────────────────────────────────────
│ True
└──────────────────────────────────────────────────────────
```

State Transition Table:

Current	Event	Guard	Next	Action	Send
START			Ready	InitInterface	
Ready	Lookup		Waiting		DoLookup
Ready	AddEntry		Waiting		DoAddEntry
Waiting	TransFound		Ready	DisplayResult	
Waiting	TransNotFound		Ready	DisplayError	

114

Appendix B. AWSOME Analysis Model for **Bā′bəl**

The following AWSOME code formally captures the **Bā′bəl** analysis model.

```
Package AnalysisModel is

    type String is Seq of Char;
    type EntrySeq is Seq of Entry;
    type Language is (English, Japanese, German);
    type PBState is (activated, deactivated);

    Class Word is
        var spelling       : String;
        var origin         : Language;
    end Class;

    Class EngWord is Word with
        invariant (origin = English);
    end Class;

    Class NihWord is Word with
        invariant (origin = Japanese);
    end Class;

    Class GerWord is Word with
        invariant (origin = German);
    end Class;

    Class Entry is
        var eng_word       : EngWord;
        var nih_word       : NihWord;
        var ger_word       : GerWord;

        invariant
            (size(eng_word.spelling) > 0);
    end Class;

    Class BabelDictionary is
        var table          : EntrySeq;

        procedure InitDictionary()
            guarantees
                size(table') = 0;

        procedure AddEntry(entry? : in Entry)
            guarantees
                entry? in table';

        procedure FindTranslation(word? : in String,
                                  tgtLang? : in Language
                                  translation! : out String)
```

```
        guarantees
            (forall e : Entry spot
                (e in table and e.eng_word.spelling = word?) implies
                (tgtLang? = Japanese and translation! = e.nih_word.spelling) or
                (tgtLang? = German and translation! = e.ger_word.spelling)));

invariant
    (forall e1, e2 : Entry; i, j : Nat spot
        (i leq size(table) and i leq j and e1 = table(i) and e2 = table(j))
        implies reduce(concat, e1) < reduce(concat, e2));

dynamic model{

    state Start;
    state Ready;
    state Busy;

    event DoLookup(aWord : String);
    event DoAddEntry(anEntry : Entry);
    event NotFound;
    event Found;
    event TransFound(result : String);
    event TransNotFound;

    transition{
        initState       Start;
        receiveEvent    InitDictionary;
        nextState       Ready;
    }
    transition{
        initState       Ready;
        receiveEvent    DoLookup;
        action          FindTranslation;
        nextState       Busy;
    }
    transition{
        initState       Ready;
        receiveEvent    DoAddEntry;
        action          AddEntry;
        nextState       Ready;
    }
    transition{
        initState       Busy;
        receiveEvent     NotFound;
        sendEvent       TransNotFound;
        nextState       Ready;
    }
    transition{
        initState       Busy;
        receiveEvent    Found;
        sendEvent       TransFound;
```

```
                  nextState        Ready;
            }
      }
end Class;

Class PushButton is
      var name          : String;
      var status        : PBState;
end Class;

Class AddEntryPButton is PushButton with

      procedure InitAddEntryPButton()
            guarantees
                  (status' = deactivated);

      procedure ToggleStatus()
            guarantees
                  ((status = deactivated implies status' = activated) and
                   (status = activated implies status' = deactivated));

      dynamic model{

            state Start;
            state Deactivated;
            state Activated;

            event ButtonPressed;
            event AddEntry;

            transition{
                  initState        Start;
                  receiveEvent     InitAddEntryPButton;
                  nextState        Deactivated;
            {
            transition{
                  initState        Deactivated;
                  receiveEvent     ButtonPressed;
                  action           ToggleStatus;
                  sendEvent        AddEntry;
                  nextState        Activated;
            }
            transition{
                  initState        Activated;
                  action           ToggleStatus;
                  nextState        Deactivated;
            }
      }
end Class;

Class LookupButton is PushButton with
```

117

```
procedure InitLookupPButton()
    guarantees
        (status' = deactivated);

procedure ToggleStatus()
    guarantees
        ((status = deactivated implies status' = activated) and
         (status = activated implies status' = deactivated));

dynamic model{

    state Start;
    state Deactivated;
    state Activated;

    event ButtonPressed;
    event Lookup;

    transition{
        initState       Start;
        receiveEvent    InitLookupPButton;
        nextState       Deactivated;
    {
    transition{
        initState       Deactivated;
        receiveEvent    ButtonPressed;
        action          ToggleStatus;
        sendEvent       Lookup;
        nextState       Activated;
    }
    transition{
        initState       Activated;
        action          ToggleStatus;
        nextState       Deactivated;
    }
    }
end Class;

Class TextBox is
    var name        : String;
    var value       : String;
    var editable    : Boolean;
end Class;

Class StatusBar is
    var name        : String;
    var value       : String;
end Class;

Class BabelInterface is
```

```
var tgtLang      : Language;
var e_word       : TextBox;
var f_word       : TextBox;
var translate    : LookupPButton;
var addEntry     : AddEntryPButton;
var sBar         : StatusBar;

procedure InitInterface()
    guarantees
        (size(e_word') = 0 and size(f_word') = 0 and sBar.value' = "Ready.");

procedure DisplayResult(result? : in String)
    guarantees
        (f_word.value' = result?);

procedure DisplayError()
    guarantees
        (sBar.value' = e_word.value + " not found in dictionary.");

invariant
    (e_word.editable = True and f_word.editable = False);

dynamic model{

    state Start;
    state Ready;
    state Waiting;

    event Lookup;
    event AddEntry;
    event TransNotFound;
    event TransFound(result : String);
    event DoLookkup(aWord : String);
    event DoAddEntry(anEntry : Entry);
    }
end Class;

end Package;
```

Appendix C. AWSOME Object-oriented Architecture for Bā′ bəl

The following AWSOME code formally captures the **Bā′ bəl** architecture model for a simple object-oriented (i.e., non-hierarchically homogeneous) architectural style.

```
Package OOArchModel_Babel is

    Class Component is
    end Class;

    Class Connector is
    end Class;

    Class Port is
    end Class;

    Class Role is
    end Class;

    Class Comp_BabelDictionary is Component with
    end Class;

    Association Comp_BabelDictionary_relates_to_BabelDictionary is
        role theComponent        : Comp_BabelDictionary;
        role theClass            : BabelDictionary;
    end Association;

    Class RcvPort_DoLookup is Port with
    end Class;

    Class RcvPort_DoAddEntry is Port with
    end Class;

    Class RcvPort_NotFound is Port with
    end Class;

    Class RcvPort_Found is Port with
    end Class;

    Class SndPort_TransNotFound is Port with
    end Class;

    Class SndPort_TransFound is Port with
    end Class;

    Aggregation Comp_BabelDictionary_has_ports is
        parent theComponent      : Comp_BabelDictionary;
        child  aPort             : RcvPort_DoLookup;
    end Aggregation;
```

```
Aggregation Comp_BabelDictionary_has_ports is
    parent theComponent              : Comp_BabelDictionary;
    child  aPort                     : RcvPort_DoAddEntry;
end Aggregation;

Aggregation Comp_BabelDictionary_has_ports is
    parent theComponent              : Comp_BabelDictionary;
    child  aPort                     : RcvPort_NotFound;
end Aggregation;

Aggregation Comp_BabelDictionary_has_ports is
    parent theComponent              : Comp_BabelDictionary;
    child  aPort                     : RcvPort_Found;
end Aggregation;

Aggregation Comp_BabelDictionary_has_ports is
    parent theComponent              : Comp_BabelDictionary;
    child  aPort                     : SndPort_TransNotFound;
end Aggregation;

Aggregation Comp_BabelDictionary_has_ports is
    parent theComponent              : Comp_BabelDictionary;
    child  aPort                     : SndPort_TransFound;
end Aggregation;

Class Comp_AddEntryPButton is Component with
end Class;

Association Comp_AddEntryPButton_relates_to_AddEntryPButton is
    role theComponent                : Comp_AddEntryPButton;
    role theClass                    : AddEntryPButton;
end Association;

Class RcvPort_ButtonPressed is Port with
end Class;

Class SndPort_AddEntry is Port with
end Class;

Aggregation Comp_AddEntryPButton_has_ports is
    parent theComponent              : Comp_AddEntryPButton;
    child  aPort                     : RcvPort_ButtonPressed;
end Aggregation;

Aggregation Comp_AddEntryPButton_has_ports is
    parent theComponent              : Comp_AddEntryPButton;
    child  aPort                     : SndPort_AddEntry;
end Aggregation;

Class Comp_LookupPButton is Component with
end Class;
```

```
Association Comp_LookupPButton_relates_to_LookupPButton is
    role theComponent           : Comp_LookupPButton;
    role theClass               : LookupPButton;
end Association;

Class SndPort_Lookup is Port with
end Class;

Aggregation Comp_LookupPButton_has_ports is
    parent theComponent         : Comp_LookupPButton;
    child  aPort                : RcvPort_ButtonPressed;
end Aggregation;

Aggregation Comp_LookupPButton_has_ports is
    parent theComponent         : Comp_LookupPButton;
    child  aPort                : SndPort_Lookup;
end Aggregation;

Class Comp_BabelInterface is Component with
end Class;

Association Comp_BabelInterface_relates_to_BabelInterface is
    role theComponent           : Comp_BabelInterface;
    role theClass               : BabelInterface;
end Association;

Class RcvPort_Lookup is Port with
end Class;

Class RcvPort_AddEntry is Port with
end Class;

Class RcvPort_TransFound is Port with
end Class;

Class RcvPort_TransNotFound is Port with
end Class;

Class SndPort_DoLookup is Port with
end Class;

Class SndPort_DoAddEntry is Port with
end Class;

Class SndPort_DisplayError is Port with
end Class;

Aggregation Comp_BabelInterface_has_ports is
    parent theComponent         : Comp_BabelInterface;
    child  aPort                : RcvPort_Lookup;
```

```
end Aggregation;

Aggregation Comp_BabelInterface_has_ports is
    parent theComponent            : Comp_BabelInterface;
    child  aPort                   : RcvPort_AddEntry;
end Aggregation;

Aggregation Comp_BabelInterface_has_ports is
    parent theComponent            : Comp_BabelInterface;
    child  aPort                   : RcvPort_TransFound;
end Aggregation;

Aggregation Comp_BabelInterface_has_ports is
    parent theComponent            : Comp_BabelInterface;
    child  aPort                   : RcvPort_TransNotFound;
end Aggregation;

Aggregation Comp_BabelInterface_has_ports is
    parent theComponent            : Comp_BabelInterface;
    child  aPort                   : SndPort_DoLookup;
end Aggregation;

Aggregation Comp_BabelInterface_has_ports is
    parent theComponent            : Comp_BabelInterface;
    child  aPort                   : SndPort_DoAddEntry;
end Aggregation;

Aggregation Comp_BabelInterface_has_ports is
    parent theComponent            : Comp_BabelInterface;
    child  aPort                   : SndPort_DisplayError;
end Aggregation;

Class Conn_TransNotFound is Connector with
end Class;

Class Conn_TransFound is Connector with
end Class;

Class Conn_AddEntry is Connector with
end Class;

Class Conn_Lookup is Connector with
end Class;

Class Conn_DoLookup is Connector with
end Class;

Class Conn_DoAddEntry is Connector with
end Class;

Class InRole is Role with
```

```
end Class;

Class OutRole is Role with
end Class;

Aggregation Conn_TransNotFound_has_roles is
    parent theConnector          : Conn_TransNotFound;
    child  aRole                 : InRole;
end Aggregation;

Aggregation Conn_TransNotFound_has_roles is
    parent theConnector          : Conn_TransNotFound;
    child  aRole                 : OutRole;
end Aggregation;

Aggregation Conn_TransFound_has_roles is
    parent theConnector          : Conn_TransFound;
    child  aRole                 : InRole;
end Aggregation;

Aggregation Conn_TransFound_has_roles is
    parent theConnector          : Conn_TransFound;
    child  aRole                 : OutRole;
end Aggregation;

Aggregation Conn_AddEntry_has_roles is
    parent theConnector          : Conn_AddEntry;
    child  aRole                 : InRole;
end Aggregation;

Aggregation Conn_AddEntry_has_roles is
    parent theConnector          : Conn_AddEntry;
    child  aRole                 : OutRole;
end Aggregation;

Aggregation Conn_Lookup_has_roles is
    parent theConnector          : Conn_Lookup;
    child  aRole                 : InRole;
end Aggregation;

Aggregation Conn_Lookup_has_roles is
    parent theConnector          : Conn_Lookup;
    child  aRole                 : OutRole;
end Aggregation;

Aggregation Conn_DoLookup_has_roles is
    parent theConnector          : Conn_DoLookup;
    child  aRole                 : InRole;
end Aggregation;

Aggregation Conn_DoLookup_has_roles is
```

```
      parent theConnector          : Conn_DoLookup;
      child  aRole                 : OutRole;
end Aggregation;

Aggregation Conn_DoAddEntry_has_roles is
      parent theConnector          : Conn_DoAddEntry;
      child  aRole                 : InRole;
end Aggregation;

Aggregation Conn_DoAddEntry_has_roles is
      parent theConnector          : Conn_DoAddEntry;
      child  aRole                 : OutRole;
end Aggregation;

Association RcvPort_DoLookup_attaches_to_OutRole is
      role thePort                 : RcvPort_DoLookup;
      role theRole                 : OutRole;
end Association;

Association RcvPort_DoAddEntry_attaches_to_OutRole is
      role thePort                 : RcvPort_DoAddEntry;
      role theRole                 : OutRole;
end Association;

Association SndPort_TransNotFound_attaches_to_InRole is
      role thePort                 : SndPort_TransNotFound;
      role theRole                 : InRole;
end Association;

Association SndPort_TransFound_attaches_to_InRole is
      role thePort                 : SndPort_TransFound;
      role theRole                 : InRole;
end Association;

Association SndPort_AddEntry_attaches_to_InRole is
      role thePort                 : SndPort_AddEntry;
      role theRole                 : InRole;
end Association;

Association SndPort_Lookup_attaches_to_InRole is
      role thePort                 : SndPort_Lookup;
      role theRole                 : InRole;
end Association;

Association RcvPort_Lookup_attaches_to_OutRole is
      role thePort                 : RcvPort_Lookup;
      role theRole                 : OutRole;
end Association;

Association RcvPort_AddEntry_attaches_to_OutRole is
      role thePort                 : RcvPort_AddEntry;
```

```
        role theRole                    : OutRole;
    end Association;

    Association RcvPort_TransFound_attaches_to_OutRole is
        role thePort                    : RcvPort_TransFound;
        role theRole                    : OutRole;
    end Association;

    Association RcvPort_TransNotFound_attaches_to_OutRole is
        role thePort                    : RcvPort_TransNotFound;
        role theRole                    : OutRole;
    end Association;

    Association SndPort_DoLookup_attaches_to_InRole is
        role thePort                    : SndPort_DoLookup;
        role theRole                    : InRole;
    end Association;

    Association SndPort_DoAddEntry_attaches_to_InRole is
        role thePort                    : SndPort_DoAddEntry;
        role theRole                    : InRole;
    end Association;

end Package;
```

Appendix D.   Acme Output for **Bā′bəl** *Object-oriented Architecture*

The following Acme code was automatically generated using the WsAcmeVisitor class. The WsAcmeVisitor class contains a visit() method for each node type contained in an AWSOME architecture model (i.e., WsPackage, WsClass, and WsAssociation). When called, each visit method examines its node to determine the purpose served by the node and outputs the appropriate Acme surface syntax. For example, a WsAssociation node can serve the purpose of an architectural *attachment* or an architectural *binding*. The visit() method keys on the name of the WsAssociation node, determines whether it is an *attachment* (i.e., WsIdentifier = "attaches_to") or a *binding* (i.e., WsIdentifier = "binds"), and responds accordingly.

```
Acme translation for AWSOME architecture model: OOArchModel_Babel Automatically
generated by WsAcmeVisitor on Feb 27, 2000 at 3:29 PM

System OOArchModel_Babel = {

    Component Comp_BabelDictionary = {

        Port RcvPort_DoLookup;

        Port RcvPort_DoAddEntry;

        Port RcvPort_NotFound;

        Port RcvPort_Found;

        Port SndPort_TransNotFound;

        Port SndPort_TransFound;

    };

    Component Comp_AddEntryPButton = {

        Port RcvPort_ButtonPressed;

        Port SndPort_AddEntry;

    };

    Component Comp_LookupPButton = {

        Port RcvPort_ButtonPressed;

        Port SndPort_Lookup;
```

127

```
};

Component Comp_BabelInterface = {

    Port RcvPort_Lookup;

    Port RcvPort_AddEntry;

    Port RcvPort_TransFound;

    Port RcvPort_TransNotFound;

    Port SndPort_DoLookup;

    Port SndPort_DoAddEntry;

    Port SndPort_DisplayError;

};

Connector Conn_TransNotFound = {

    Role InRole;

    Role OutRole;

};


Connector Conn_TransFound = {

    Role InRole;

    Role OutRole;

};


Connector Conn_AddEntry = {

    Role InRole;

    Role OutRole;

};


Connector Conn_Lookup = {

    Role InRole;
```

```
    Role OutRole;

};


Connector Conn_DoLookup = {

    Role InRole;

    Role OutRole;

};


Connector Conn_DoAddEntry = {

    Role InRole;

    Role OutRole;

};


Attachments {

    Comp_BabelDictionary.RcvPort_DoLookup to Conn_DoLookup.OutRole

    Comp_BabelDictionary.RcvPort_DoAddEntry to Conn_DoAddEntry.OutRole

    Comp_BabelDictionary.SndPort_TransNotFound to Conn_TransNotFound.InRole

    Comp_BabelDictionary.SndPort_TransFound to Conn_TransFound.InRole

    Comp_AddEntryPButton.SndPort_AddEntry to Conn_AddEntry.InRole

    Comp_LookupPButton.SndPort_Lookup to Conn_Lookup.InRole

    Comp_BabelInterface.RcvPort_Lookup to Conn_Lookup.OutRole

    Comp_BabelInterface.RcvPort_AddEntry to Conn_AddEntry.OutRole

    Comp_BabelInterface.RcvPort_TransFound to Conn_TransFound.OutRole

    Comp_BabelInterface.RcvPort_TransNotFound to Conn_TransNotFound.OutRole

    Comp_BabelInterface.SndPort_DoLookup to Conn_DoLookup.InRole

    Comp_BabelInterface.SndPort_DoAddEntry to Conn_DoAddEntry.InRole

};
```

The following AWSOME code formally captures the **Bā' bəl** architecture model for a client-server, object-oriented (i.e., hierarchically heterogeneous) architectural style.

```
Package ArchModel_Babel_CS is

    Class Component is
    end Class;

    Class ComponentType_Object is Component with
    end Class;

    Class ComponentType_Client is Component with
    end Class;

    Class ComponentType_Server is Component with
    end Class;

    Class Connector is
    end Class;

    Class ConnectorType_MethodCall is Connector with
    end Class;

    Class ConnectorType_RPC is Connector with
    end Class;

    Class Port is
    end Class;

    Class PortType_MethodCall is Port with
    end Class;

    Class PortType_RPC is Port with
    end Class;

    Class Role is
    end Class;

    Class RoleType_MethodCall is Role with
    end Class;

    Class RoleType_RPC is Role with
    end Class;

    Class Style is
    end Class;

    Class StyleType_ClientServer is Style with
```

130

```
end Class;

Class StyleType_ObjectOriented is Style with
end Class;

Package Configuration is

    Class Style is StyleType_ClientServer with
    end Class;

    Class SndPort_RPC is PortType_RPC with
    end Class;

    Class RcvPort_RPC is PortType_RPC with
    end Class;

    Class Comp_BabelClient is ComponentType_Client with
    end Class;

    Aggregation Comp_BabelClient_has_ports is
        parent theComponent          : Comp_BabelClient;
        child  aPort                 : SndPort_RPC;
    end Aggregation;

    Aggregation Comp_BabelClient_has_ports is
        parent theComponent          : Comp_BabelClient;
        child  aPort                 : RcvPort_RPC;
    end Aggregation;

    Class Comp_BabelServer is ComponentType_Server with
    end Class;

    Aggregation Comp_BabelServer_has_ports is
        parent theComponent          : Comp_BabelServer;
        child  aPort                 : SndPort_RPC;
    end Aggregation;

    Aggregation Comp_BabelServer_has_ports is
        parent theComponent          : Comp_BabelServer;
        child  aPort                 : RcvPort_RPC;
    end Aggregation;

    Class Conn_RPC is ConnectorType_RPC with
    end Class;

    Class InRole_Client is RoleType_RPC with
    end Class;

    Class OutRole_Client is RoleType_RPC with
    end Class;
```

```
Class InRole_Server is RoleType_RPC with
end Class;

Class OutRole_Server is RoleType_RPC with
end Class;

Aggregation Conn_RPC_has_role is
    parent theConnector          : Conn_RPC;
    child  aRole                 : InRole_Client;
end Association;

Aggregation Conn_RPC_has_role is
    parent theConnector          : Conn_RPC;
    child  aRole                 : OutRole_Client;
end Association;

Aggregation Conn_RPC_has_role is
    parent theConnector          : Conn_RPC;
    child  aRole                 : InRole_Server;
end Association;

Aggregation Conn_RPC_has_role is
    parent theConnector          : Conn_RPC;
    child  aRole                 : OutRole_Server;
end Association;

Association SndPort_RPC_attaches_to_InRole_Client is
    role thePort                 : SndPort_RPC;
    role theRole                 : InRole_Client;
end Association;

Association RcvPort_RPC_attaches_to_OutRole_Client is
    role thePort                 : RcvPort_RPC;
    role theRole                 : OutRole_Client;
end Association;

Association SndPort_RPC_attaches_to_InRole_Server is
    role thePort                 : SndPort_RPC;
    role theRole                 : InRole_Server;
end Association;

Association RcvPort_RPC_attaches_to_OutRole_Server is
    role thePort                 : RcvPort_RPC;
    role theRole                 : OutRole_Server;
end Association;

Package Comp_BabelClient is

    Class Style is StyleType_ObjectOriented with
    end Class;
```

```
Class Comp_AddEntryPButton is ComponentType_Object with
end Class;

Class RcvPort_AddEntryPressed is PortType_MethodCall with
end Class;

Class SndPort_AddEntry is PortType_MethodCall with
end Class;

Aggregation Comp_AddEntryPButton_has_ports is
    parent theComponent      : Comp_AddEntryPButton;
    child  aPort             : RcvPort_AddEntryPressed;
end Aggregation;

Aggregation Comp_AddEntryPButton_has_ports is
    parent theComponent      : Comp_AddEntryPButton;
    child  aPort             : SndPort_AddEntry;
end Aggregation;

Class Comp_LookupPButton is ComponentType_Object with
end Class;

Class RcvPort_LookupPressed is PortType_MethodCall with
end Class;

Class SndPort_Lookup is PortType_MethodCall with
end Class;

Aggregation Comp_LookupPButton_has_ports is
    parent theComponent      : Comp_LookupPButton;
    child  aPort             : RcvPort_LookupPressed;
end Aggregation;

Aggregation Comp_LookupPButton_has_ports is
    parent theComponent      : Comp_LookupPButton;
    child  aPort             : SndPort_Lookup;
end Aggregation;

Class BabelInterface is ComponentType_Object with
end Class;

Class RcvPort_Lookup is PortType_MethodCall with
end Class;

Class RcvPort_AddEntry is PortType_MethodCall with
end Class;

Class RcvPort_TransFound is PortType_MethodCall with
end Class;

Class RcvPort_TransNotFound is PortType_MethodCall with
```

```
end Class;

Class SndPort_DoLookup is PortType_MethodCall with
end Class;

Class SndPort_DoAddEntry is PortType_MethodCall with
end Class;

Class SndPort_DisplayError is PortType_MethodCall with
end Class;

Aggregation has_port is
     parent theComponent       : Comp_BabelInterface;
     child  aPort              : RcvPort_Lookup;
end Aggregation;

Aggregation has_port is
     parent theComponent       : Comp_BabelInterface;
     child  aPort              : RcvPort_AddEntry;
end Aggregation;

Aggregation has_port is
     parent theComponent       : Comp_BabelInterface;
     child  aPort              : RcvPort_TransFound;
end Aggregation;

Aggregation has_port is
     parent theComponent       : Comp_BabelInterface;
     child  aPort              : RcvPort_TransNotFound;
end Aggregation;

Aggregation has_port is
     parent theComponent       : Comp_BabelInterface;
     child  aPort              : RcvPort_DoLookup;
end Aggregation;

Aggregation has_port is
     parent theComponent       : Comp_BabelInterface;
     child  aPort              : RcvPort_DoAddEntry;
end Aggregation;

Aggregation has_port is
     parent theComponent       : Comp_BabelInterface;
     child  aPort              : RcvPort_DisplayError;
end Aggregation;

Class InRole is RoleType_MethodCall with
end Class;

Class OutRole is RoleType_MethodCall with
end Class;
```

```
Class Conn_AddEntry is ConnectorType_MethodCall with
end Class;

Aggregation Conn_AddEntry_has_roles is
    parent theConnector      : Conn_AddEntry;
    child  aRole             : InRole;
end Aggregation;

Aggregation Conn_AddEntry_has_roles is
    parent theConnector      : Conn_AddEntry;
    child  aRole             : OutRole;
end Aggregation;

Class Conn_Lookup is ConnectorType_MethodCall with
end Class;

Aggregation Conn_Lookup_has_roles is
    parent theConnector      : Conn_Lookup;
    child  aRole             : InRole;
end Aggregation;

Aggregation Conn_Lookup_has_roles is
    parent theConnector      : Conn_Lookup;
    child  aRole             : OutRole;
end Aggregation;

Association SndPort_AddEntry_attaches_to_InRole is
    role thePort             : SndPort_AddEntry;
    role theRole             : InRole;
end Association;

Association RcvPort_AddEntry_attaches_to_OutRole is
    role thePort             : RcvPort_AddEntry;
    role theRole             : OutRole;
end Association;

Association SndPort_Lookup_attaches_to_InRole is
    role thePort             : SndPort_Lookup;
    role theRole             : InRole;
end Association;

Association RcvPort_Lookup_attaches_to_OutRole is
    role thePort             : RcvPort_Lookup;
    role theRole             : OutRole;
end Association;

Association SndPort_RPC_binds_to_SndPort_DoLookup is
    role theOuterPort        : SndPort_RPC;
    role theInnerPort        : SndPort_DoLookup;
end Association;
```

```
    Association SndPort_RPC_binds_to_SndPort_DoAddEntry is
        role theOuterPort        : SndPort_RPC;
        role theInnerPort        : SndPort_DoAddEntry;
    end Association;

    Association RcvPort_RPC_binds_to_RcvPort_TransFound is
        role theOuterPort        : RcvPort_RPC;
        role theInnerPort        : RcvPort_TransFound;
    end Association;

    Association RcvPort_RPC_binds_to_RcvPort_TransNotFound is
        role theOuterPort        : RcvPort_RPC;
        role theInnerPort        : RcvPort_TransNotFound;
    end Association;

end Package;

Package Comp_BabelServer is

    Class Style is StyleType_ObjectOriented with
    end Class;

    Class Comp_BabelDictionary is ComponentType_Object with
    end Class;

    Class RcvPort_DoLookup is PortType_MethodCall with
    end Class;

    Class RcvPort_DoAddEntry is PortType_MethodCall with
    end Class;

    Class SndPort_TransFound is PortType_MethodCall with
    end Class;

    Class SndPort_TransNotFound is PortType_MethodCall with
    end Class;

    Aggregation Comp_BabelDictionary_has_ports is
        parent theComponent      : Comp_BabelDictionary;
        child  aPort             : RcvPort_DoLookup;
    end Aggregation;

    Aggregation Comp_BabelDictionary_has_ports is
        parent theComponent      : Comp_BabelDictionary;
        child  aPort             : RcvPort_DoAddEntry;
    end Aggregation;

    Aggregation Comp_BabelDictionary_has_ports is
        parent theComponent      : Comp_BabelDictionary;
        child  aPort             : RcvPort_TransFound;
```

```
        end Aggregation;

        Aggregation Comp_BabelDictionary_has_ports is
            parent theComponent        : Comp_BabelDictionary;
            child   aPort              : RcvPort_TransNotFound;
        end Aggregation;

        Association RcvPort_RPC_binds_to_RcvPort_DoLookup is
            role theOuterPort          : RcvPort_RPC;
            role theInnerPort          : RcvPort_DoLookup;
        end Association;

        Association RcvPort_RPC_binds_to_RcvPort_DoAddEntry is
            role theOuterPort          : RcvPort_RPC;
            role theInnerPort          : RcvPort_DoAddEntry;
        end Association;

        Association SndPort_RPC_binds_to_SndPort_TransFound is
            role theOuterPort          : SndPort_RPC;
            role theInnerPort          : SndPort_TransFound;
        end Association;

        Association SndPort_RPC_binds_to_SndPort_TransNotFound is
            role theOuterPort          : SndPort_RPC;
            role theInnerPort          : SndPort_TransNotFound;
        end Association;

    end Package;

  end Package;

end Package;
```

Appendix F. Acme Output for **Bā′ bəl** *Client-Server Architecture*

The following Acme code was automatically generated using the WsAcmeVisitor class. The WsAcmeVisitor class contains a visit() method for each node type contained in an AWSOME architecture model (i.e., WsPackage, WsClass, and WsAssociation). When called, each visit method examines its node to determine the purpose served by the node and outputs the appropriate Acme surface syntax. For example, a WsClass node can serve the purpose of an architectural *component* or an architectural *connector*. The visit() method keys on the name of the WsClass node, determines whether it is a *component* (i.e., WsIdentifier begins with "Comp_") or a *connector* (i.e., WsIdentifier begins with "Conn_"), and responds accordingly.

```
Acme translation for AWSOME architecture model: ArchModel_Babel_CS
Automatically generated by WsAcmeVisitor on Feb 27, 2000 at 3:30 PM

System ArchModel_Babel_CS = {

   Component Comp_BabelClient = {

      Representation = {

         System Comp_BabelClient_details = {

            Component Comp_AddEntryPButton = {

               Port RcvPort_AddEntryPressed;

               Port SndPort_AddEntry;

            };

            Component Comp_LookupPButton = {

               Port RcvPort_LookupPressed;

               Port SndPort_Lookup;

            };

            Component Comp_BabelInterface = {

               Port RcvPort_Lookup;

               Port RcvPort_AddEntry;
```

```
        Port RcvPort_TransFound;

        Port RcvPort_TransNotFound;

        Port SndPort_DoLookup;

        Port SndPort_DoAddEntry;

        Port SndPort_DisplayError;

    };

    Connector Conn_AddEntry = {

        Role InRole;

        Role OutRole;

    };

    Connector Conn_Lookup = {

        Role InRole;

        Role OutRole;

    };

    Attachments {

        Comp_AddEntryPButton.SndPort_AddEntry to Conn_AddEntry.InRole

        Comp_BabelInterface.RcvPort_AddEntry to Conn_AddEntry.OutRole

        Comp_LookupPButton.SndPort_Lookup to Conn_Lookup.InRole

        Comp_BabelInterface.RcvPort_Lookup to Conn_Lookup.OutRole

    };

};

Bindings {

    SndPort_RPC to Comp_BabelInterface.SndPort_DoLookup;
```

```
            SndPort_RPC to Comp_BabelInterface.SndPort_DoAddEntry;

            RcvPort_RPC to Comp_BabelInterface.RcvPort_TransFound;

            RcvPort_RPC to Comp_BabelInterface.RcvPort_TransNotFound;

        };

    };

};

Component Comp_BabelServer = {

    Representation = {

        System Comp_BabelServer_details = {

            Component Comp_BabelDictionary = {

                Port RcvPort_DoLookup;

                Port RcvPort_DoAddEntry;

                Port SndPort_TransFound;

                Port SndPort_TransNotFound;

            };

        };

        Bindings {

            RcvPort_RPC to Comp_BabelDictionary.RcvPort_DoLookup;

            RcvPort_RPC to Comp_BabelDictionary.RcvPort_DoAddEntry;

            SndPort_RPC to Comp_BabelDictionary.SndPort_TransFound;

            SndPort_RPC to Comp_BabelDictionary.SndPort_TransNotFound;

        };

    };
```

```
};

Connector Conn_RPC = {

    Role InRole_Client;

    Role OutRole_Server;

    Role OutRole_Client;

    Role InRole_Server;

};

Attachments {

    Comp_BabelClient.SndPort_RPC to Conn_RPC.InRole_Client

    Comp_BabelClient.RcvPort_RPC to Conn_RPC.OutRole_Client

    Comp_BabelServer.SndPort_RPC to Conn_RPC.InRole_Server

    Comp_BabelServer.RcvPort_RPC to Conn_RPC.OutRole_Server

};

};
```

Bibliography

1. Alexander, Christopher, et al. *A Pattern Language*. New York: Oxford University Press, 1977.

2. Allen, Robert. *A Formal Approach to Software Architecure*. PhD dissertation, Carnegie Mellon University, Pittsburgh, PA 15213, May 1997. CMU-CS-97-144.

3. Anderson, Gary L. *An Interactive Tool for Refining Software Specifications from a Formal Domain Model*. MS thesis, Air Force Institute of Technology, Wright–Patterson AFB, OH, March 1999. DTIC Number ADA361745.

4. Balzer, Robert, et al. "Software Technology in the 1990's: Using a New Paradigm," *Computer*, *16*(11):39–45 (November 1983).

5. Bass, Len, et al. *Software Architecture in Practice*. Reading, Massachusetts: Addison–Wesley, 1998.

6. Bowen, Jonathan P. and Michael G. Hinchey. "Seven More Myths of Formal Methods," *IEEE Software*, 34–41 (July 1995).

7. Bowen, Jonathan P. and Michael G. Hinchey. "Ten Commandments of Formal Methods," *IEEE Computer*, *28*(4):56–62 (April 1995).

8. Coplien, James O. and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*. Reading, Massachusetts: Addison–Wesley, 1995.

9. Cornn, Gary L. *A Software Synthesis System Based on an Object-Oriented Repository*. MS thesis, Air Force Institute of Technology, Wright–Patterson AFB, OH, March 2000. AFIT/GCS/ENG/00M-05.

10. Frederick P. Brooks, Jr. *The Mythical Man-Month* (Anniversary Edition). Reading, Massachusetts: Addison–Wesley, 1995.

11. Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison–Wesley, 1995.

12. Garlan, David, et al. "Exploiting Style in Architectural Design Environments." *Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press, December 1994.

13. Garlan, David, et al. "Acme: An Architecture Description Interchange Language." *Proceedings of CASCON'97*. 169–183. November 1997.

14. Graham, Robert P. *Lecture Notes, Part 1: CSCE 793—Formal Methods in Software Engineering*. Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio, March 1999.

15. Grassman, Winfried Karl and Jean-Paul Tremblay. *Logic and Discrete Mathematics: A Computer Science Perspective*. Upper Saddle River, New Jersey: Prentice Hall, 1996.

16. Hartrum, Thomas C. "An Object Oriented Formal Transformation System for Primitive Object Classes." Air Force Institute of Technology, June 1999.

17. Jacobson, Ivar, et al. *The Unified Software Development Process*. Reading, Massachusetts: Addison–Wesley, 1999.

18. Jones, Cliff B. *Recent Books on Formal Methods*. Technical Report, Department of Computer Science: The University of Manchester, 1995.

19. Jones, Cliff B. "A Rigorous Approach to Formal Methods," *IEEE Computer*, *29*(4) (April 1996).

20. Marsh, David W. *Formal Object State Model Transformations for Automated Agent System Synthesis*. MS thesis, Air Force Institute of Technology, Wright–Patterson AFB, OH, March 2000. AFIT/GCE/ENG/00M-03.

21. Monroe, Robert T., et al. "Architectural Styles, Design Patterns, and Objects," *IEEE Software*, *14*(1):43–52 (January 1997).

22. Noe, Penelope A. *A Structured Approach to Software Tool Integration*. MS thesis, Air Force Institute of Technology, Wright–Patterson AFB, OH, March 1999. DTIC Number ADA361674.

23. Pressman, Roger S. *Software Engineering: A Practitioner's Approach* (Third Edition). New York, New York: McGraw-Hill, 1992.

24. Pressman, Roger S. *Software Engineering: A Practitioner's Approach* (Fourth Edition). New York, New York: McGraw-Hill, 1997.

25. Robert P. Graham, Jr. *Common Object-oriented Imperative Language: Language Reference Manual*. Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio, September 1999.

26. Robinson, David J. *A Component Based Approach to Agent Specification*. MS thesis, Air Force Institute of Technology, Wright–Patterson AFB, OH, March 2000. AFIT/GCS/ENG/00M-22.

27. Rumbaugh, James, et al. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice–Hall, 1991.

28. Shaw, Mary and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, New Jersey: Prentice–Hall, 1996.

29. Shock, Robert C. "Software Architecture Course Notes." Department of Computer Science and Engineering, Wright State University, 1999.

30. Sommerville, Ian. *Software Engineering* (Fifth Edition). Harlow, England: Addison–Wesley, 1996.

31. Spivey, J.M. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge: Cambridge University Press, 1988.

32. Woodcock, Jim and Jim Davies. *Using Z: Specification, Refinement, and Proof*. London: Printice Hall, 1996.

33. Woodcock, Jim and Martin Loomes. *Software Engineering Mathematics*. London: Pitman, 1988.

34. Wordsworth, J.B. *Software Development with Z*. Wokingham, England: Addision–Wesley, 1992.

Vita

Captain Darin L. Williams was born on 28 February 1966 in Houston, Texas. He graduated from Lufkin High School in Lufkin, Texas in 1984. He enlisted in 1986 and served over eight years on active duty as a Communications–Computer Systems Operator in the United States Air Force. During his enlistment, he entered undergraduate studies at the University of Maryland Asian Division where he graduated with a Bachelor of Science degree in Computer Studies in 1994. In the same year, he was commissioned through the Air Force Officer Training School where he was recognized as a Distinguished Graduate and nominated for a Regular Commission.

His first assignment as a commissioned officer was to the Air Force Pentagon Communications Agency where he served as a Software Project Manager and full-time member of the Software Engineering Process Group. In August 1998, he relocated his family to Dayton, Ohio and entered the Graduate School of Engineering and Management, Air Force Institute of Technology. Upon graduation, he will be assigned to the Standard Systems Group, Maxwell AFB/Gunter Annex, Alabama. He is married to Mariko Takano of Nagano City, Japan and has two children: Zachary Makoto and Ariel Reina.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2000	Master's Thesis

4. TITLE AND SUBTITLE
EXPLICITLY MODELING HIERARCHICALLY HETEROGENEOUS SOFTWARE ARCHITECTURES IN AN OBJECT-ORIENTED FORMAL TRANSFORMATION SYSTEM

5. FUNDING NUMBERS

6. AUTHOR(S)
Darin L. Williams, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 P Streen, Building 640
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/00M-25

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFRL/IFTD
Attn: Roy F. Stratton
525 Brooks Rd
Rome, NY 13441-4505
(330)315-3004 DSN:587-3004

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Maj Robert P. Graham, Jr., ENG, DSN:785-3636 ext. 4595, robert.graham@afit.af.mil

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Formal software transformation systems are software development environments typified by the semi-automated application of a series of correctness-preserving transformations to formal data models. The range of software architectures such systems are capable of producing is often restricted by the limited ability to accept high-level design inputs as constraints on the transformation process. When architectural inputs are acceptable, often the modeling language excludes the explicit representation of architectural constructs and provides, at best, an extremely limited architectural analysis capability. This research defines a high-level taxonomy of software architectures and proposes a way to explicitly model a broad class of architectures by adapting the native object-oriented modeling language to the task. Using the AFIT Wide-Spectrum Object-Modeling Environment (AWSOME) as a proving ground, it demonstrates the ability to fully automate the transformation of an object-oriented analysis model to a non-hierarchically homogeneous, object-oriented architecture. Additionally, it demonstrates the ability to explicitly model the richer class of hierarchically heterogeneous software architectures in an object-oriented transformation system and to gain insight into the behavioral characteristics of such architectures by exporting them to an architectural interchange language for external analysis.

14. SUBJECT TERMS
Software Architecture, Transformation System, Formal Methods, Object-Oriented Design, Hierarchically Heterogeneous Software Architecture, Architecture Taxonomy, Architecture Classification, Software Synthesis, Architectural Interchange

15. NUMBER OF PAGES
159

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to *stay within the lines* to meet *optical scanning requirements.*

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C - Contract   PR - Project
G - Grant   TA - Task
PE - Program   WU - Work Unit
    Element   Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with....; Trans. of....; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE - See authorities.
NASA - See Handbook NHB 2200.2.
NTIS - Leave blank.

Block 12b. Distribution Code.

DOD - Leave blank.
DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA - Leave blank.
NTIS - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.